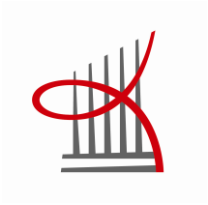


Entiteettirakenne ja -hierarkia peliohjelmoinnissa

Mikael Juhala

Opinnäytetyö
Joulukuu 2013
Tietotekniikka
Ohjelmistotekniikka

TAMPEREEN AMMATTIKORKEAKOULU
Tampere University of Applied Sciences

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietotekniikka
Ohjelmistotekniikka

MIKAEL JUHALA:
Entiteettihierarkia ja -rakenne peliohjelmoinnissa

Opinnäytetyö 42 sivua
Joulukuu 2013

Opinnäytetyössä käydään läpi erinäisiä aihealueita peliohjelmointiin liittyen, keskittyen entiteettien toteutukseen ja käyttöön. Työ on jaettu pääasiassa kolmeen osaan, jotka kaikki keskittyvät yhteen kokonaisuuteen. Ensimmäisessä osiossa esitellään kaksi hierarkiamallia, joilla entiteettien väliset suhteet voi määritellä, toisessa osiossa esitellään neljä eri tapaa toteuttaa yksittäiset entiteetit ja kolmannessa keskitytään viestintämenetelmiin, joilla tietoa voidaan välittää sekä pelimoottorissa että itse pelissä.

Jokaisesta toteutustavasta käydään korkealla tasolla läpi niiden teoria ja annetaan muutamia esimerkkejä mahdollisista käyttökohteista, joissa esiin tuotuja asioita voisi soveltaa.

Kaksi ensimmäistä osiota etenee samaa kaavaa mukaillen, jossa ensin esitellään yksi toteutustapa ja tuodaan esiin sen vahvuudet. Sitten toteutustavasta tuodaan ilmi siihen liittyvät ongelmakohdat, joita seuraavan luvun esittelemän toteutustavan on tarkoitus korjata tai kiertää.

Opinnäytetyön käsittelemiä aiheita on tutkittu pääasiassa hyödyntämällä opittuja asioita sekä harrasteprojekteissa että pelialan työtehtävissä, mutta myös opiskelemalla alaan liittyvää materiaalia. Opinnäytetyön lopussa annetaan tekijän omia pohdintoja ja selvitetään, miten eri aiheet oli valittu, miten niitä oli hyödynnetty käytännössä ja mihin lopputulokseen päädyttiin.

Asiasanat: pelimoottori, entiteetti, hierarkia, viestintä

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
ICT Engineering
Software Engineering

MIKAEL JUHALA:
Entity hierarchy and structure in game programming

Bachelor's thesis 42 pages
December 2013

This thesis goes through different subjects related to game programming, focusing on the implementation and use of entities. The thesis is divided into three parts, each concentrated on one subject. The first part introduces two hierarchy models, which are used to define the relations between entities. The second part showcases four different ways to implement single entities and the third part focuses on methods of messaging that are used to convey information within the game engine and the game itself.

Every implementation has its theory explained on a high level. Each chapter also gives a few examples on practical uses where the introduced things could be applied.

The two first parts more or less follow the same pattern. First they introduce one implementation and explain its strengths. However, at the end of a chapter they show problems observed in the given examples. Then the next chapter will introduce another implementation, which is supposed to either fix or circumvent the problems.

The subjects of the thesis have been researched by mainly applying the learned things in both hobby projects and work assignments in game industry, but also by studying material related to the subject. At the end of the thesis are presented the writer's own speculations and clarified on how the subjects were chosen, how they were used in practice and what was the final conclusion.

Key words: game engine, entity, hierarchy, messaging

SISÄLLYS

ERITYISSANASTO

deserialisointi	Tiedon muuntamista tiedostoista tai muistipuskurista sovelluksen tietorakenteiksi. Vastakohtana serialisointi, jossa muunnos tehdään vastakkaiseen suuntaan.
entiteetti	Entiteetti on peliohjelmoinnissa jokin pelimaailman konkreettinen asia, esimerkiksi hahmo tai nappi. Niitä kutsutaan usein myös peliobjekteiksi.
kompositio	Kompositiossa jonkin luokan ilmentymä sisältää toisen luokan ilmentymän ja hallitsee tämän elinkaarta. Kun omistaja tuhotaan, tuhotaan myös sen sisältämät ilmentymät.
metodin otsikko	Metodin otsikko koostuu metodin paluuarvosta ja sen vastaanottamista parametrien tietotyypeistä.
muunnosmatriisi	Matriisi, joka sisältää entiteetin sijainnin, kulman ja skaalan joko 2- tai 3-ulotteisessa koordinaatistossa.
MVC	Model-View-Controller, arkkitehtuurimalli, jossa vastuu on jaettu kolmen eri osan kesken.
MVP	Model-View-Presenter, MVC:n muunnelma.
overhead	Tehtävän suorittamiseen sisältyvä ylimääräinen työ, joka ei suoraan edistä tavoitteeseen pääsyä. Voi tarkoittaa sovelluksessa tiedon prosessointiaikaa tai muistinkäyttöä, tai myös kehittäjältä vaadittuja ylimääräisiä tehtäviä.
pääsilmukka	Silmukka, josta päivitys- ja piirtokutsut ensisijaisesti lähetetään. Peli voi käytännössä koostua useammastakin silmukasta.

1 JOHDANTO

Toteutin syksyn 2012 aikana yksinkertaisen pelimoottorin ensin Applen iOS-käyttöjärjestelmälle, jonka jälkeen käänsin sen myös Microsoft Windows 7:lle. Vuoden 2013 aikana pelimoottorin viimeisin iteraatio toteutettiin Applen OS X -käyttöjärjestelmälle (versio 10.9, ”Mavericks”) C#-kielellä MonoMac-kirjastoa hyödyntäen. Idea omalle pelimoottorille tuli tutustuessa peliohjelmointiin Java-ohjelmointikielellä Android-käyttöjärjestelmälle (tuolloin versio 2.2.x). Tarkoituksena ei niinkään ollut tehdä kaupalliseen käyttöön suunnattua pelimoottoria, vaan muodostaa siitä harrastus, jota koulun ja työn ohessa kehittämällä voi oppia lisää ohjelmoinnista.

Pelimoottori sisältää ominaisuuksia, joiden koetaan olevan pelimoottorien perustoimintoja. Nämä sisältävät kuvien renderöinnin, pelikohtaukset, entiteettien hierarkian ja rakenteen, viestijärjestelmän, resurssienhallinnan erilaisten tiedostojen lataamiseen ja käyttöön, liikkeen animoinnin, sekä entiteettien deserialisoinnin tekstitiedostoista. Pelimoottori on toteutettu useassa eri iteraatiossa, joissa jokaisessa on pyritty löytämään erilaisia tapoja toteuttaa pelimoottorin arkkitehtuuri. Opinnäytetyössä tuodaan ilmi niiden vahvuuksia ja heikkouksia, mutta myös muita niistä opittuja asioita.

Opinnäytetyössä käydään läpi kaksi entiteettihierarkiaa ja neljä rakennetta, joilla yksittäiset entiteetit voidaan toteuttaa. Ensin käydään läpi jokaisen toteutustavan teoria ja vahvuudet sekä annetaan niistä muutama käytännön esimerkki. Lopuksi tuodaan esiin niistä aiheutuvia ongelmia, joita aina seuraavan luvun toteutustavan on tarkoitus korjata tai kiertää. Tarkoituksena on siis edetä omia havaintoja ja lähdemateriaalia mukaillen huonoimmasta toteutustavasta parhaimpaan. Nämä aiheet käydään kuitenkin läpi melko korkealla tasolla, eli tavoitteena ei ole antaa tarkkoja ohjeita eri rakenteiden toteuttamiseen.

Opinnäytetyö käsittää myös katsauksen eri viestintämenetelmiin, joita entiteetit voivat pelimoottorissa käyttää tiedon välittämiseen. Tämä osio ei noudata samaa kaavaa entiteettihierarkioiden ja –rakenteiden kanssa. Viestintämenetelmiä voi ja on hyväkin käyttää vapaasti keskenään, valiten aina käytännöllisimmän menetelmän tapauskohtaisesti. Näistä etenkin takaisinkutsut ja keskitetty viestijärjestelmä käydään läpi hieman matalammalla tasolla.

Tämän opinnäytetyön päätelmät perustuvat pääasiassa omiin käytännön kokemuksiin ja havaintoihin, joita on verrattu sekä kollegoiden suosittelemiin että luotettavina pidettyihin lähdemateriaaleihin, kuten Tampereen teknillisen yliopiston professorien kirjoihin sekä Addison-Wesley-sarjan kirjoihin, jotka ovat alalla yhä suosittuja iästään huolimatta. Monien termien osalta on käytetty lähdemateriaalina tietojenkäsittelytieteen laitoksen sivuilta löytyvää Harri Laineen (Helsingin yliopisto, 1996) kirjoittamaa sanastoa. Mikäli termille ei ole yleisesti käytössä olevaa suomenkielistä termiä, on niistä käytetty englanninkielestä peräisin olevia vastineita, jotka on selitetty lyhyesti opinnäytetyön erikoissanastossa.

Jokaisesta entiteettirakenteesta ja -hierarkiasta annetaan esimerkkejä UML-kaavioina ja tarvittaessa myös ohjelmakoodina. UML-kaaviot on toteutettu Gliffy-websovelluksella ja niissä noudatetaan vuoden 2004 standardia, eli versiota 2.0. Ohjelmakoodit on kirjoitettu C#-kielen vuonna 2010 julkaistulla 4.0 versiolla (käytössä .NET:n versio 4.0). Käytetty ohjelmointikieli on valittu sen syntaksin ja rakenteen vuoksi, joka muistuttaa monia muita laajalti käytettyjä ohjelmointikieliä, kuten C++:aa ja Javaa. Valinnalla on pyritty varmistamaan ohjelmakoodin ymmärrettävyys lukijan taustasta riippumatta.

2 PELIMOOTTORI YLEISESTI

Pelimoottorilla tarkoitetaan ohjelmistoa, jonka päälle pelejä voidaan toteuttaa. Pelimoottorin tarkoituksena on tarjota pelin käyttöön erinäisiä kirjastoja esimerkiksi äänien ja musiikin soittamiseen, kuvien renderöimiseen, animointiin, 2D- ja 3D-matematiikkaan, käyttäjän syötteen lukemiseen, verkkoliikenteeseen, fysiikoihin ja pelimaailman rakentamiseen. Pelimoottori ei itsessään sisällä pelin logiikkaa, vaan tarjoaa ainoastaan kehitysympäristön sen toteuttamiseen. (Ward 2008; Sivak 2009.)

Raja pelin ja pelimoottorin välillä on kuitenkin nykyään hämärtynyt, mikä johtuu useimmiten tarkoituksesta, jonka pelimoottorin halutaan täyttävän. Jotkin pelimoottorit on suunniteltu juuri tietynlaisten pelien tekemiseen ja sen eri osat on toteutettu tätä tarkoitusta varten. Käytännössä tämä tarkoittaa sitä, että pelimoottori tekee olettamuksia itse pelistä ja saattaa toteuttaa osan sen logiikasta. Tällaisia pelimoottoreita on kuitenkin hankala hyödyntää edelleen täysin erilaisissa peleissä, mutta niiden tarkoituksena onkin luoda mahdollisimman optimaalinen kehitysympäristö. Mitä yleiskäyttöisempi pelimoottorista tehdään, sitä enemmän joudutaan tekemään kompromisseja. (Gregory 2009, 11-13.)

Esimerkkejä tällaisista tiettyihin pelityyppeihin kohdistetuista pelimoottoreista on *Adventure Game Studio* (Chris Jones, 1997), jolla nimensä mukaisesti on tarkoitus tehdä seikkailupelejä, sekä *Source* (Valve Corporation, 2004), joka on kehitetty varta vasten FPS-pelien (*First Person Shooter*) kehittämiseen.

3 PELIKOHTAUKSEN RAKENNE

Sovellukset koostuvat usein monista eri ruuduista, aktiviteeteista tai ikkunoista, mutta peliohjelmoinnissa niitä kutsutaan usein pelikohtauksiksi. Pelikohtaus on jokin konkreettinen osa peliä, joka voidaan eriyttää muusta pelistä. Esimerkiksi pelin päävalikot voivat olla yksi pelikohtaus ja pelimaailman eri alueet voivat kukin olla oma pelikohtauksensa. Siirtyminen kohtauksesta toiseen tarkoittaa edellisen kohtauksen tuhoamista ja uuden lataamista muistiin. Entiteetit ja niiden välinen hierarkia luodaan kohtausta ladattaessa.

Jokaisella pelikohtauksella on oma entiteettihierarkiansa ja mahdollisesti myös oma logiikkansa, jolla pelikohtauksen tapahtumat etenevät. Pelikohtaukset ovat toisistaan eristettyjä, mikä tarkoittaa sitä, että viestejä ei välitetä suoraan pelikohtauksien välillä. Yleensä tällaisesta viestinvälityksestä vastaa ylempi luokka, joka omistaa pelikohtaukset ja hallitsee siirtymiä niiden välillä. Edellinen pelikohtaus voi esimerkiksi luoda väliaikaisen tietueen, joka sisältää tietoja pelikohtauksen tapahtumista. Tämä yläluokka tallentaa tietueen ja tarjoaa uudelle pelikohtaukselle lukuoikeuden tietueeseen, mikäli se on tarpeen. Entiteettejä voidaan kuitenkin tarvittaessa tuoda edellisestä pelikohtauksesta uuteen, mikäli niiden toiminta ei saa katketa pelikohtauksien vaihdon aikana tai niiden lataamia resursseja tarvitaan molemmissa pelikohtauksissa, eikä näiden resurssien vapauttaminen ja lataaminen uudelleen olisi käytännöllistä. Esimerkkinä entiteetti, joka vastaa pelin taustamusiikin soittamisesta. Usein taustamusiikin halutaan sulavasti vaihtuvan ääniraidasta toiseen pelikohtausta vaihdettaessa, jolloin kyseistä entiteettiä ei voida tuhota ja sen resursseja vapauttaa. (Zechner & Green 2011, 97-103.)

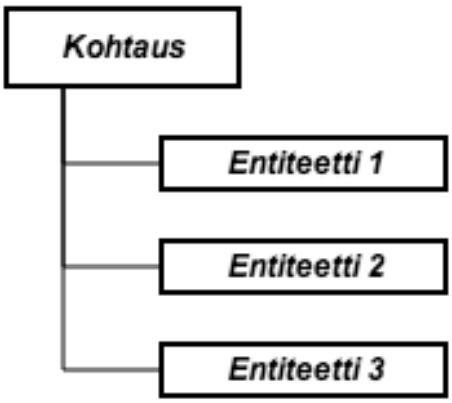
Pelikohtauksen entiteettihierarkia on erityisen riippuvainen itse entiteettien toteutuksesta, sillä pelin logiikka voidaan toteuttaa joko entiteeteillä tai täysin näistä erillään. Vastaavasti entiteettien toteutus on riippuvainen valitusta hierarkiasta, sillä eri hierarkioissa luokkien vastuualueet ovat erilaiset. Tästä syystä hierarkia on hyvä valita jo pelimoottorin toteutuksen alussa.

Seuraavissa luvuissa on käyty läpi kaksi erilaista hierarkiaa ja tuotu esiin niiden vahvuuksia ja heikkouksia, sekä mahdollisia keinoja heikkouksien kiertämiseen.

Hierarkioiden lisäksi luvussa 3.3 esitellään ns. globaalit järjestelmät, jota voi soveltaa kumpaankin näistä hierarkioista. Niiden käyttö ei suoranaisesti vaikuta entiteettien rakenteeseen tai hierarkiaan, vaan siirtävät vastuuta entiteeteistä hierarkian ulkopuolelle. Jokaisesta hierarkiasta on annettu ohjelmakoodiesimerkkejä niiden pääsilmukoiden toteuttamiseen, sekä UML-kaaviot havainnollistamaan rakenteita. Pääsilmukalla tarkoitetaan peliohjelmoinnissa silmukkaa, jota suoritetaan, kunnes peli suljetaan. Yleensä se laskee kulunutta aikaa, päivittää pelimaailman ja lopulta renderöi sen ruudulle. Pelimoottorista ja sen arkkitehtuurista riippuen silmukka voi olla vastuussa myös muista tehtävistä. Joillain alustoilla, kuten Apple iOS:n Cocoa-kehitysympäristössä, pääsilmukka on piilotettu kehittäjältä ja alusta on vastuussa päivitys- ja piirtokutsujen lähettämisestä sovellukselle. Seuraavissa luvuissa keskitytään kuitenkin yksinkertaiseen alustasta eristettyyn *while*-silmukkaan. (Lord 2012).

3.1 Lineaarinen hierarkia

Lineaarisessa hierarkiassa entiteetit ovat kaikki samalla tasolla ja ns. samanarvoisia. Tämä ei kuitenkaan tarkoita, että ne välttämättä olisivat saman olion omistuksessa. Pelillä voi erikseen olla luokat esimerkiksi pelihahmojen ja ympäristön luomiseen ja omistamiseen, mutta niiden entiteetit ovat silti hierarkiassa samanarvoisia. Hierarkiaa on havainnollistettu kuvassa 1.



KUVA 1: Lineaarinen hierarkia

Samanarvoisuudella entiteettihierarkiassa tarkoitetaan sitä, että yhden entiteetin muokkaaminen ei välttämättä vaikuta muihin, eikä entiteettien välillä ole yleisesti

määriteltyjä omistajuussuhteita tai vastuujakoa. Pääsilmukan toteuttaminen tällaiselle hierarkialle on yksinkertaista. Esimerkki tällaisesta on annettu ohjelmakoodissa 1.

```
bool running = true;
float time = getCurrentTime();

while ( running )
{
    float newTime = getCurrentTime();
    float delta = newTime - time;

    foreach ( Opponent op in m_opponents )
    {
        op.updateAi( delta );
    }

    Input input = readInput();
    m_player.updatePlayer( delta, input );

    foreach ( Entity e in m_allEntities )
    {
        e.render();
    }

    running = ( getCoinsLeft() > 0 );
    time = newTime;
}
```

OHJELMAKOODI 1: Lineaarisen hierarkian pääsilmukka

Ohjelmakoodin 1 silmukka voidaan sisällyttää esimerkiksi pelikohtaukseen. Tällöin jokainen pelikohtaus voi suorittaa myös kohtausriippuvaista logiikkaa päivityskutsujen välissä. Pääsilmukassa on lista pelin vastustajista, pelaaja sekä lista kaikista entiteeteistä koko pelikohtauksessa. Ensin pääsilmukka päivittää vastustajien tekoälyn, sitten luetaan syöte ja päivitetään pelaajaa vastaavan entiteetin tila sen pohjalta. Pääsilmukan lopussa kaikki entiteetit renderöidään ruudulle. Tätä silmukkaa suoritetaan niin kauan kuin pelin lopetusehdot eivät täyty. Tässä tapauksessa pääsilmukka tarkistaa, montako kolikkoa pelissä on jäljellä. Kun silmukka päättyy, voi peli joko sammua tai siirtyä toiseen pelikohtaukseen.

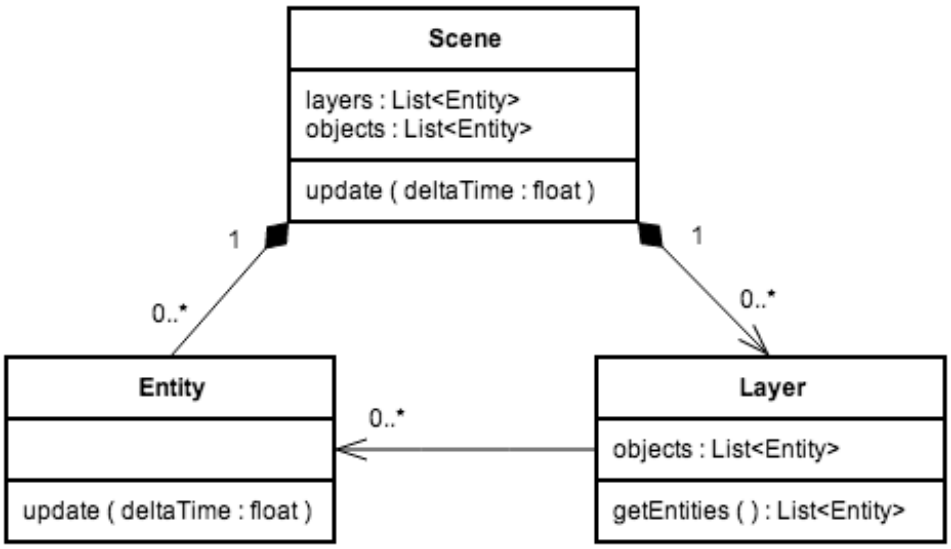
Tällaisen pääsilmukan käyttö on todella suoraviivaista ja mahdollisten virheiden löytäminen on yksinkertaisissa peleissä helppoa, sillä asioiden päivitysjärjestys on täysin ennalta määritetty. Ongelmia kuitenkin ilmenee, kun peliä halutaan laajentaa tai olemassa olevia ominaisuuksia muokata. Esimerkkinä tilanne, jossa entiteetti tarvitsee oman tilansa päivittämiseen toisen tietyn entiteetin päivitettyä tilaa. Tällöin jälkimmäisenä mainittu entiteetti on päivitettävä ensin.

Ohjelmakoodissa 2 on kyseistä ongelma pyritty ratkaisemaan kerroksittaisella päivittämisellä. Sama esitetty myös UML-kaaviossa 1. Pelikohtaus siis sisältää listan kerroksista, joista jokainen sisältää viitteet tietynlaisiin entiteetteihin, jotka pelikohtaus on kerroksia luodessa niille antanut. Pääsilmukassa kerrokset päivitetään yksitellen. Näin pelikohtaus voidaan päivittää pienemmissä osissa.

```
foreach ( Layer layer in m_layers )
{
    List<Entity> entities = layer.getEntities();

    foreach ( Entity e in entities )
    {
        e.update( delta );
    }
}
```

OHJELMAKOODI 2: Entiteettien jakaminen kerroksiin



KAAVIO 1: Entiteettien jakaminen kerroksiin

Päivityskerrokset kuitenkin rajoittavat yhtä lailla pelin laajennettavuutta. Mikäli vain yksittäistä entiteettiä, eikä kokonaista tietotyyppiä, halutaan päivittää erikseen, on sitä varten luotava uusi kerros tai päivitettävä se erillään kerroksista. Kun taas samanlaisillekin entiteeteille on useampia kerroksia, on uusien entiteettien lisääminen dynaamisesti hankalaa, sillä entiteetin toiminta voi olla riippuvainen kerroksien ja entiteettien päivitysjärjestyksestä.

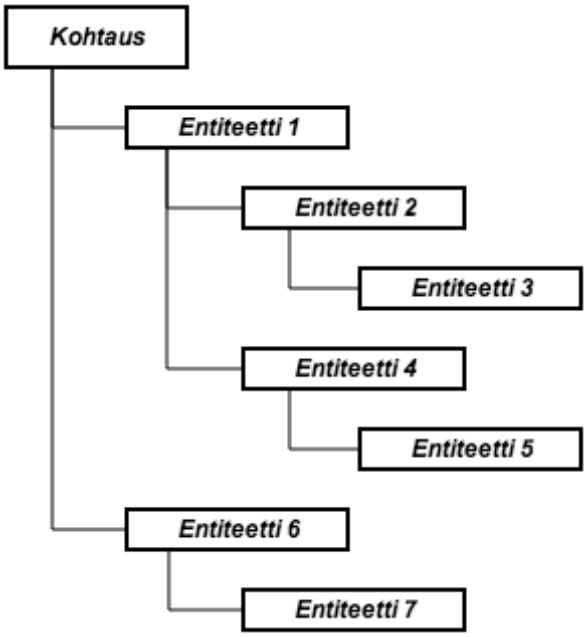
Ongelmia tulee myös monimutkaisempien rakenteiden mallintamisessa, koska entiteeteillä ei ole hierarkisia suhteita keskenään. Esimerkkinä entiteetti, jonka on tarkoitus mallintaa autoa, jonka moottori halutaan kuitenkin mallintaa erillisenä entiteetteinä, jolla on oma logiikkansa. Ilman hierarkian määrittelemää omistajuussuhdetta on moottorin omistajuus hankalaa antaa autolle. Pelikohtauksen näkökulmasta nämä entiteetit olisivat kaksi erillistä entiteettiä, joita on päivitettävä ja hallittava erikseen.

Ammattimaisissa pelimoottoreissa ei tällaista hierarkiaa käytetä, mutta se on edelleen suosittu aloittelevien peliohjelmoijien keskuudessa. Monet pelkästään pelien tekemiseen tarkoitetut ohjelmat, kuten *Game Maker*, rajoittavatkin rakenteen tällaiseksi, jotta se olisi mahdollisimman helppo sisäistää.

3.2 Puuhierarkia

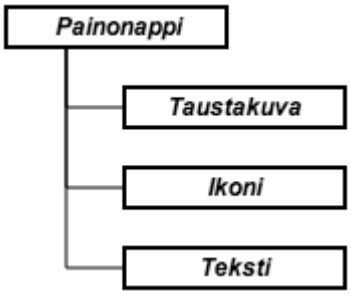
Puuhierarkiassa entiteeteillä on tarkkaan määritelty hierarkia, jossa ne ovat toistensa ylä- tai alientiteettejä (kutsutaan myös englanniksi *parent*- ja *child*-entiteeteiksi). Ylemmän entiteetin muokkaaminen yleensä vaikuttaa tavalla tai toisella kaikkiin sen suoriin alientiteetteihin, sekä niistä alaspäin aina hierarkian pohjalle saakka. Suoralla alientiteetillä tarkoitetaan entiteettiä, joka on puuhierarkiassa yhden kerroksen toisen entiteetin alapuolella.

Esimerkkinä kuvan 2 osoittama hierarkia, jossa entiteetit 2 ja 4 ovat entiteetin 1 suoria alientiteettejä. Myös entiteetti 7 on yhden kerroksen 1:tä alempana, mutta se sijaitsee puuhierarkian eri haarassa, jolloin niiden välillä ei ole hierarkista suhdetta. Puuhierarkiassa samalla tasolla olevat entiteetit, joilla on sama yläentiteetti, ovat toistensa sisarentiteettejä. Entiteetit ovat toisiensa sisarentiteettejä myös silloin, kun millään niistä ei ole yläentiteettiä määritettynä ollenkaan.



KUVA 2: Puuhierarkia

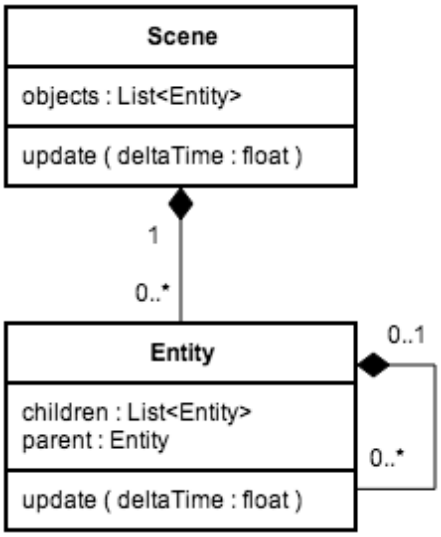
Tavallisimpia tietoja, joiden muokkaaminen vaikuttaa muihin, ovat entiteettien muunnosmatriisi ja aktiivisuus, eli onko entiteetti asetettu päälle vai pois. Puuhierarkiassa entiteeteillä on kompositiosuhde toisiinsa, eli yläentiteetin tuhoaminen aiheuttaa kaikkien sen alientiteettien tuhoamisen. Muita mahdollisia alientiteetteihin vaikuttavia asioita ovat esimerkiksi näkyvyys kamerassa, fysiikkamallinnus ja törmäystunnistukset. Koska yläentiteetin tilan muutos vaikuttaa alientiteetteihin, suunnitellaan puuhierarkiat siten, että joka haaran entiteetit koostavat tiettyyn konkreettiseen kokonaisuuden. Otetaan esimerkiksi painonappi, jota vastaa yksi entiteetti, ja jonka alientiteetteinä ovat sen eri osat, kuten taustakuva, ikoni ja teksti. Kun painonapin ylimmäistä entiteettiä muokataan, vaikuttavat muutokset myös alientiteetteihin. Tätä hierarkiaa on havainnollistettu kuvassa 3. Vastaavasti painonappi voisi olla käyttöliittymää kuvaavan entiteetin alientiteetti.



KUVA 3: Yksinkertaisen painonapin hierarkia

Puuhierarkiaa alettiinkin käyttää juuri tämänkaltaisten entiteettien välisten suhteiden toteuttamiseen. Näin vältetään sisällyttämästä liikaa ominaisuuksia yhteen entiteettiin, vaan ne voidaan jakaa osiin; taustakuva ja ikoni kumpikin ovat vastuussa vain yhden kuvan piirtämisestä, teksti on vastuussa fontin piirtämisestä ja juurientiteettinä toimiva painonappi lukee ja käsittelee käyttäjän syötettä.

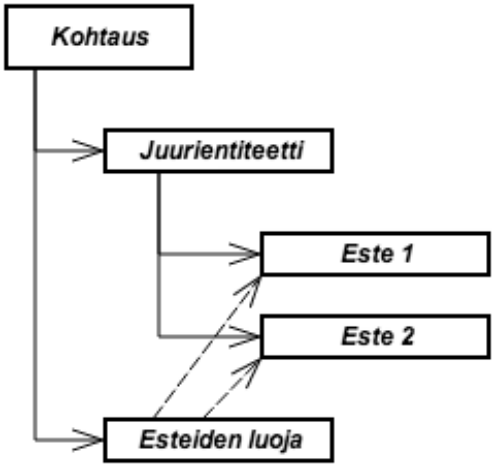
Yksinkertaisin tapa toteuttaa puuhierarkia on antaa omistajuus entiteeteistä niiden yläentiteeteille ja vaatimalla, että jokaisella pelikohtauksella on yksi juurientiteetti. Juurientiteeteillä ei ole yläentiteettejä, ainoastaan ali- tai sisarentiteettejä. Entiteetit ovat myös täysin vastuussa niiden suorien alientiteettien hallinnasta. Tällaisessa mallissa päivitys- ja piirtokutsut lähetetään ensin juurientiteetille, joka välittää kutsun omille alientiteeteilleen, jotka vastaavasti jatkavat välittämistä omille alientiteeteilleen. Tätä rakennetta on havainnollistettu kaaviossa 2. Kaavion osoittamassa mallissa Scene-luokka sisältää listan entiteeteistä ja päivitysmetodin. Tämä metodi kutsuu jokaisen entiteetin päivitysmetodia, jotka vastaavasti sisältävät listat omista alientiteeteistään ja ovat vastuussa näiden päivittämisestä.



KAAVIO 2: Puuhierarkia

Puuhierarkiassa omistajuussuhteet voivat muuttua joko suoraan tai epäsuoraan. Suora omistajuussuhteen muutos tapahtuu, kun entiteetin yläentiteettiä vaihdetaan. Tällöin tämä uusi yläentiteetti omistaa ko. entiteetin ja on myös vastuussa päivitys- ja piirtokutsujen välittämisestä sille. Epäsuora omistajuussuhteen muutos tapahtuu, mikäli entiteetillä, jonka yläentiteettiä vaihdetaan, on alientiteettejä. Näille hierarkian muutos ei näy suoraan.

Entiteettien välillä voi olla myös piilotettuja omistajuussuhteita. Esimerkkinä tilanne, jossa hierarkiassa on entiteetti, jonka vastuulla on luoda pelimaailmaan esteitä, mutta esteet haluttaisiin käytännöllisistä syistä asettaa hierarkiassa saman juurientiteetin alle. Esteitä luova entiteetti ei kuitenkaan välttämättä itse sijaitse hierarkiassa saman juurientiteetin alla. Hierarkian toteutuksen mukaan tuo juurientiteetti omistaa esteet, mutta ne luoneella entiteetillä voi pelin logiikan kannalta olla tarve hallita esteiden elinkaarta, mikä asettaisi sen niiden omistajaksi. Tätä tilannetta on havainnollistettu kuvassa 3. Tällaiset tilanteet aiheuttavat puuhierarkiassa helposti väärinkäsityksiä, kun kehittäjät tekevät vääriä olettamuksia entiteettien toiminnasta. Varmaa keinoa tämän ongelman kiertämiseen ei ole, mutta kattavalla ja tarkalla dokumentaatiolla ongelmia voidaan yrittää välttää.



KUVA 4: Piilotetut omistajuussuhteet puuhierarkiassa

Monimutkaisempien rakenteiden suoraviivaistamisen lisäksi puuhierarkian pääsilmukka voidaan tiivistää muutamaan riviin, kuten ohjelmakoodista 3 voidaan todeta, sillä pelikohtauksen tarvitsee päivittää ainoastaan hierarkian juurientiteetti (*m_rootEntity*), joka on vastuussa omien alientiteettiensä päivittämisestä, jotka vastaavasti ovat vastuussa omista alientiteeteistään. Päivityskutsu menee näin koko hierarkian läpi.

```
bool running = true;
float time = getCurrentTime();

while ( running )
{
    float newTime = getCurrentTime();
    float delta = newTime - time;

    m_rootEntity->update( delta );

    time = newTime;
}
```

OHJELMAKOODI 3: Puuhierarkian pääsilmukka

Puuhierarkia on kaupallisten pelimoottorien keskuudessa käytetyin rakenne. Tunnetut pelimoottorit kuten *Unity 3D* ja *Unreal Engine* hyödyntävät tätä rakennetta lähes kaikessa.

3.3 Globaalit järjestelmät

Lineaarisen hierarkian ja puuhierarkian lisänä voidaan käyttää globaaleja järjestelmiä. Ne ovat olennainen osa esimerkiksi ECS-mallia (*Entity-Component-System*), mutta niitä voi hyödyntää myös muissa malleissa. Globaali järjestelmä on luokka, jonka vastuulla on hoitaa tietynlaisten entiteettien päivittäminen. Globaalien järjestelmien tarkoitus on siirtää toimintoja entiteeteistä pois ja hallita näitä toimintoja keskitetysti.

Yksi käytännöllinen käyttökohde järjestelmille on käyttöliittymien toteutus. Kun käyttöliittymät rakennetaan erillisistä pienistä osista, tulee vastaan päivitysjärjestykseen liittyviä ongelmia. Hyvänä esimerkkinä voidaan pitää entiteettien ankkurointia ja skaalausta. Ankkuroinnilla tarkoitetaan entiteetin paikan määrittämistä suhteessa johonkin toiseen entiteettiin ja skaalauksella entiteetin koon muuttamista joko absoluuttisilla tai suhteellisilla arvoilla. Ankkuroinnin hoitava luokka valitsee kohteen, joko vapaasti määritettävä piste pelimaailmassa tai jonkin toisen entiteetin kulma tai keskipiste. Ankkurille voidaan antaa myös suhteellinen siirto kohteeseen nähden. Skaalauksesta vastaava luokka sen sijaan määrittelee kohteen, jonka suhteen entiteettiä skaalataan.

Kun taas entiteetti halutaan sekä skaalata että ankkuroida, ei ole yhtä oikeaa päivitysjärjestystä, vaan siitä riippuen entiteetin sijainti voi muuttua, sillä skaalatun

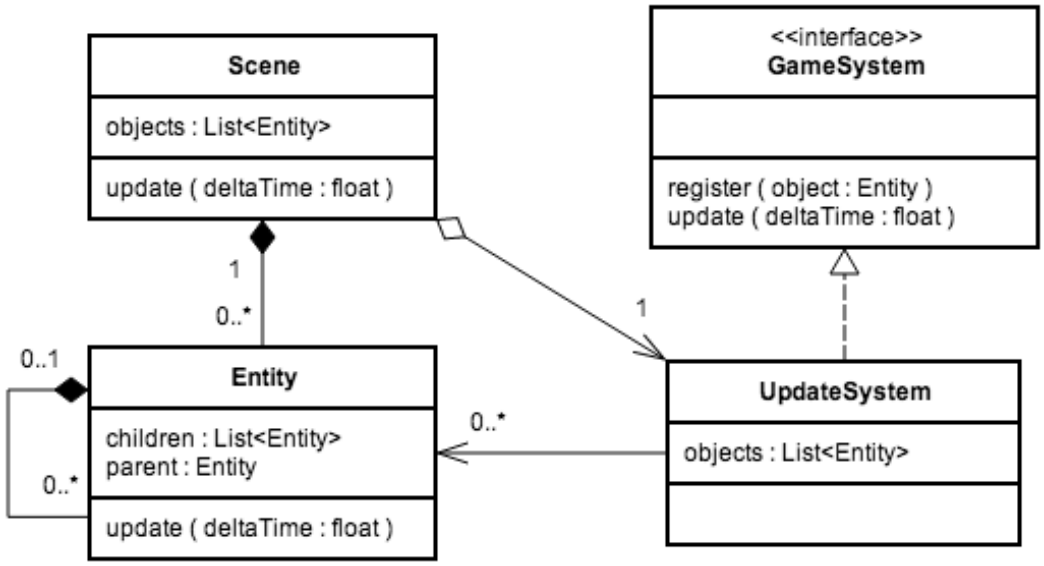
entiteetin kulma ei ole enää samassa pisteessä ruudulla kuin ennen skaalausta. Tällaisessa tilanteessa globaali keskitetty järjestelmä on hyödyllinen. Näin järjestelmälle siirretään vastuu ristiriitaisten tilanteiden ratkaisemisesta.

Globaalit järjestelmät eroavat kaaviossa 1 kuvatuista kerroksista siten, että globaalit järjestelmät voivat toimia joko pelikohtauksen ohjeistamana, vastaamalla entiteettien lähettämiin viesteihin tai täysin itsenäisesti omassa säikeessään. Kerrokset ainoastaan välittävät viestejä, eivätkä itse keskustele entiteettien kanssa.

Ongelmana globaaleissa järjestelmissä on kuitenkin vastuunjaon hämärtyminen. Esimerkiksi kuvan piirtämisen voi suorittaa joko entiteetti tai globaali järjestelmä. Ohjeistuksen määritteleminen vastuunjaosta on hankalaa, mistä syystä sama toiminto voisi loogisesti kuulua useampaan eri luokkaan.

Yksinkertaisin tapa toteuttaa globaalit järjestelmät on luoda rajapinta, jonka jokainen järjestelmä toteuttaa. Rajapinta sisältää tärkeimmät toiminnot, kuten entiteettien rekisteröimisen järjestelmään ja järjestelmän päivittämisen. Järjestelmät ovat pelikohtauksen omistuksessa, joskaan eivät välttämättä sen instantioimia. Tätä mallia on kuvattu kaaviossa 3. Monien järjestelmien kohdalla, kuten esimerkiksi äänien soittamiseen ja resurssien hallintaan keskittyvien, on kuitenkin tarpeen voida käyttää samaa järjestelmää useammassa pelikohtauksessa. Tästä syystä järjestelmät voi luoda jokin ylemmän kerroksen luokka, joka tarvittaessa antaa järjestelmiä pelikohtauksien käyttöön.

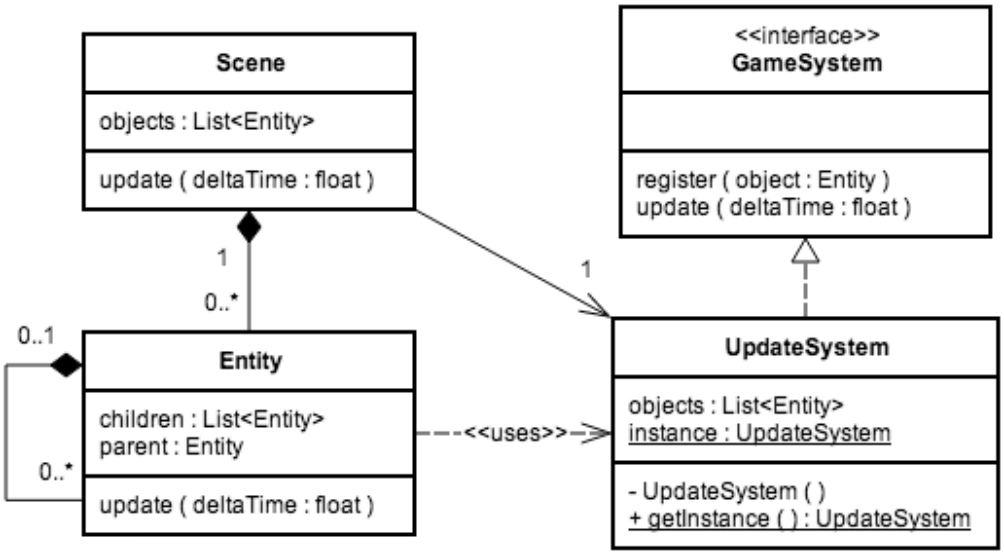
Kaaviossa 3 on kuvattu UpdateSystem, joka toteuttaa GameSystem-rajapinnan, tarjoten näin pelikohtaukselle tavan päivittää entiteettejä näennäisesti yhdellä metodikutsulla UpdateSystem-luokan ilmentymään. GameSystem-rajapinta sisältää myös *register*-metodin, jolla entiteetti on rekisteröitävä järjestelmään. Ainoastaan rekisteröidyt entiteetit päivitetään, jolloin vältetään turhat metodikutsut.



KAAVIO 3: Globaalit järjestelmät puuhierarkiassa

Kaaviossa 3 esitetyssä mallissa kuitenkin huomataan ongelma, joka ei täysin sovellu puuhierarkiaan. Entiteettien rekisteröimisen järjestelmiin nimittäin hoitaa itse pelikohtaus. Tämä soveltuu hyvin lineaariseen hierarkiaan, jossa pelikohtaus tietää kaikki sen entiteetit. Puuhierarkiassa tämä kuitenkin vaatisi järjestelmien lähettämistä kerroksissa alaspäin entiteettien rakentajissa, mikä hankaloittaa rakenteen ymmärrettävyyttä.

Puuhierarkiaa varten globaalit järjestelmät voidaan toteuttaa Singleton-mallia hyödyntäen, joka takaa, että järjestelmästä on aina vain yksi ilmentymä ja siihen on pääsy kaikkialta. Yksi ilmentymä takaa, että järjestelmän vastuualueeseen kuuluvat toiminnot hoidetaan keskitetysti. Tällöin on ongelmien etsiminen ja korjaaminen helpompaa, sillä usean ilmentymän tiloissa voi olla eroja, mikä vastaavasti voi aiheuttaa eroja niiden toimintaan. Tätä mallia on kuvattu kaaviossa 4, jossa *UpdateSystem* noudattaa Singleton-mallia. Ensimmäinen kutsu sen *getInstance*-metodiin luo ja palauttaa ilmentymän, tätä seuraavat kutsut ainoastaan palauttavat jo aiemmin luodun ilmentymän. Tällöin entiteetit voivat hoitaa rekisteröitymisen järjestelmiin itse.



KAAVIO 4: Globaalit järjestelmät Singleton-mallia hyödyntäen

Kaavion 4 kuvaamassa mallissa on kuitenkin vielä yksi ongelma. Jos järjestelmiä on useita ja niiden on kutsuttava tiettyjä metodeja entiteeteistä, on jokainen näistä toteuttava entiteetin julkiseen rajapintaan, vaikka ne eivät kaikki käyttäisikään järjestelmiä. Tämä ongelma voidaan kiertää toteuttamalla rekisteröinti takaisinkutsuilla, eli entiteetin viitteen sijaan järjestelmille lähetetään takaisinkutsuosoittimet niihin liittyviin metodeihin. Takaisinkutsujen käytöstä on kerrotaan lisää luvussa 5.2.

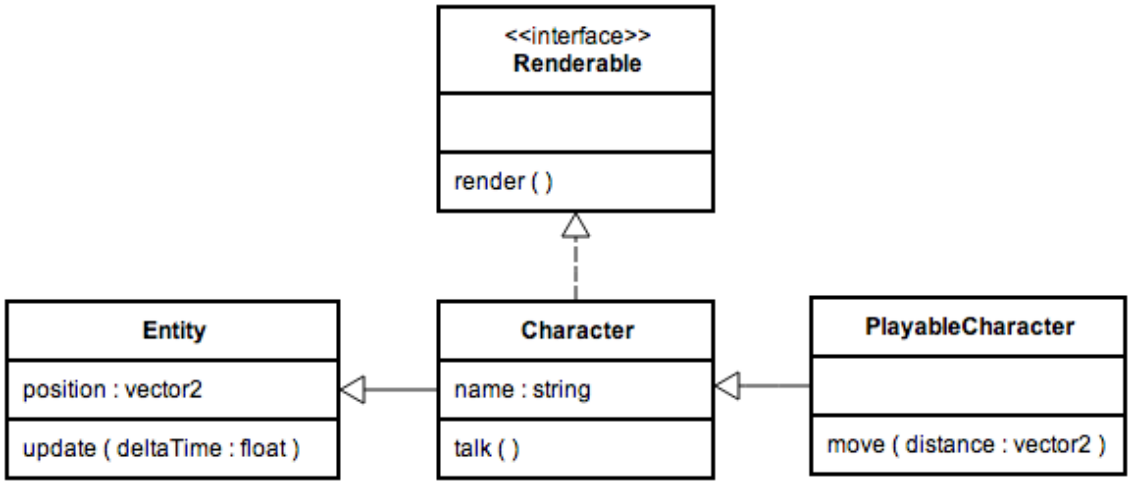
4 ENTITEETTIRAKENNE

Entiteetti vastaa pelimoottorissa yleensä yhtä konkreettista asiaa, sisältäen joko itsessään asiaan liittyvät tiedot ja toiminnot tai koostaen ne muiden luokkien ilmentymillä. Esimerkiksi pelin päähahmoa voi vastata yksi entiteetti, joka sisältää hahmon nimen, sijainnin, sen hetkisen toiminnon jne. Entiteetti on pelimoottorin keskeisin osa, sillä pelien toteutuksessa käytettävät suunnittelu- ja arkkitehtuurimallit ovat sen rakenteesta riippuvaisia. Tämä johtuu siitä, että entiteetit arkkitehtuurista riippuen joko toteuttavat itse tai sisällyttävät itseensä suurimman osan pelin logiikasta. (West 2007.)

4.1 Periyttämisketjut

Yksinkertaisin, mutta pidemmällä tähtäimellä hankalin, tapa toteuttaa entiteetit ja niiden toiminnot on toteuttaa ne yksinomaan periyttämisketjuilla, joissa jokainen ketjun luokka tuo entiteettiin lisää ominaisuuksia. Luokat voivat sisältää ominaisuuksia, jotka ovat toisistaan täysin riippumattomia. Pelin monimutkaisuudesta riippuen tällainen rakenne voi kuitenkin johtaa pitkiin ja vaikeaselkoisiin periyttämisketjuihin.

Monet aloittelevat peliohjelmoijat suosivat tällaista rakennetta, sillä se on helppo sisäistää. Käsitteet esitetään abstraktimmasta konkreettisimpaan etenevinä periyttämisketjuina. Kaaviossa 5 on esitetty tilanne, jossa abstraktimpana luokkana on Entity. Se voi sisältää vain jokaiseen entiteettiin liittyviä tietoja, kuten sijainnin. Kun tarvitaan tarkempaa tietoa, luodaan uusi luokka, joka perii edellistä. Kaaviossa Entity-luokasta on periytetty Character-luokka, joka lisää entiteettiin hahmon nimen ja kyvyn puhua. Se myös toteuttaa Renderable-rajapinnan, jonka *render*-metodilla hahmo voidaan renderöidä ruudulle. Viimeisenä periyttämisketjussa on PlayableCharacter, joka kertoo, että hahmo on pelaajan ohjattavissa ja tarvitsee tätä varten *move*-metodin, jolla hahmon sijaintia voi suoraan muokata. (West 2007; Eberly 2004, 105-106.)

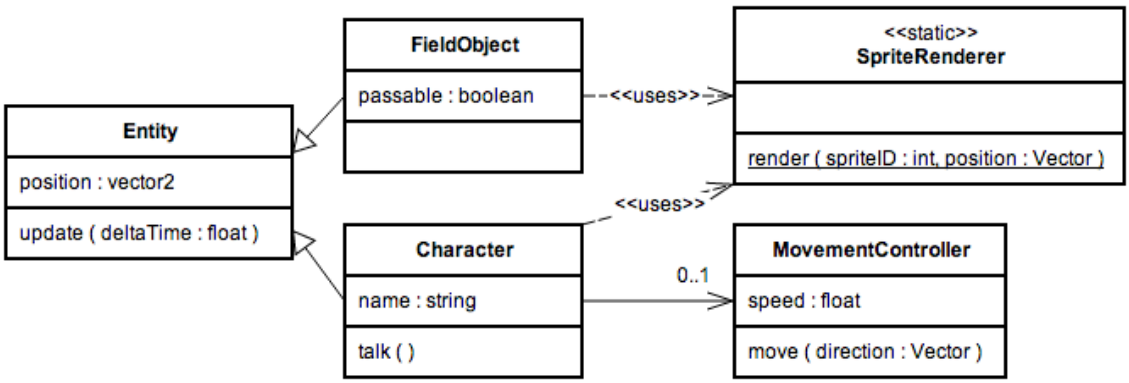


KAAVIO 5: Erikoistaminen perimällä

Periyttämisketjut voivat myös haarautua, jolloin saadaan monimutkaisiakin ketjuja. Ketjuissa voidaan käyttää hyväksi rajapintoja ja globaaleja järjestelmiä, joihin voidaan sisällyttää mahdollisesti eristettävissä olevia ominaisuuksia. Tällaiset ketjut on helppo sisäistää ja niissä vastuujako pysyy selkeänä.

Rakenne ei kuitenkaan sovellu hyvin puuhierarkiaan, sillä puuhierarkiassa olennaista on, että tiedetään hierarkian jokaisen solmukohdan eli entiteetin tarkka tietotyyppi. Reflektioita hyödyntävissä kielissä tämä ei ole ongelma, sillä oikea tietotyyppi on aina saatavilla. Kielissä, joissa reflektiota ei ole saatavilla, tämä ei kuitenkaan onnistu niin helposti, sillä tyyppimuunnoksissa kohdetyyppi on oltava tiedossa jo käännösvaiheessa.

Tyyppimuunnoksien lisäksi ongelmia ilmenee myös laajennettavuudessa. Otetaan kaavion 5 tilanne, mutta ilman globaaleja järjestelmiä. Tällöin rajapintojen toteuttaminen ei ole käytännöllistä ominaisuuksien sisällyttämiseksi entiteetteihin, sillä tämä aiheuttaa logiikan replikointia. Periyttämisketjut myös sisällyttävät luokkiin ominaisuuksia, joita ne eivät välttämättä saisi tukea. Tällöin on metodeista tehtävä joko virtuaalisia tai luotava uusi haara periyttämisketjuun. Periyttämisketjuja käytettäessä ovat keinot kuitenkin rajalliset. Kaaviossa 6 on esitetty, miten ominaisuuksia voi ulkoistaa staattisiin apuluokkiin ja paikallisesti instantioitaviin entiteetteihin. (Lord 2012.)



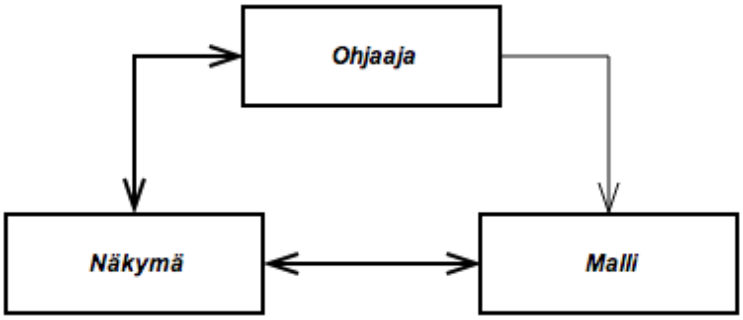
KAAVIO 6: Ominaisuuksien ulkoistaminen entiteeteistä

Kaavion 6 osoittamassa mallissa on kaksi luokkaa, FieldObject ja Character, joista kumpikin on periytetty Entity-luokasta. Ne ovat molemmat visuaalisia, eli toteuttavat jonkinlaisen näkymän, mutta sisältävät eri tiedot, ja vain Character-luokka instantioi MovementController-luokan, jota se käyttää syötteen lukemiseen ja sijaintinsa muuttamiseen. Periyttäminen FieldObject- ja Character-luokan välillä ei ole loogista, sillä niiden on tarkoitus kuvata täysin erilaisia konkreettisia asioita pelissä. Kaaviossa 6 näkymä onkin toteutettu staattisen apuluokan avulla, joka sisältää kuvan renderöimiseen tarvittavan metodin. Näin piirtämiseen vaadittava logiikka on toteutettava vain kerran ja sen laajentaminen on helppoa.

Vaikka periyttämisketjut ovatkin suosittuja pääasiassa aloittelevien ohjelmoijien keskuudessa, ajaudutaan niiden käyttöön usein myös ammattimaisissakin ohjelmistoissa. Syitä tähän on monia: refaktoroinnin ja korjauksien viivästyminen, ohjelmakoodia ei suunniteltu uudelleenkäytettäväksi tai tietyn tyyppisen kontrollirakenteen (eng. *control flow*) tai arkkitehtuurin suosiminen. Kun luokan rajapinta sisältää useita toisistaan riippumattomia metodeja, olisi luokka hyvä jakaa useampaan eri luokkaan.

4.2 Model-View-Controller

Model-View-Controller eli MVC-malli on vanhimpia arkkitehtuurimalleja. Sillä pyritään jakamaan vastuualueita eri luokille. MVC-mallissa on kolme eri osaa: ohjaaja (*controller*), malli (*model*) ja näkymä (*view*). Mallia on havainnollistettu kuvassa 5.



KUVA 5: Model-View-Controller

MVC-mallin eri osien vastuualueet ovat seuraavat:

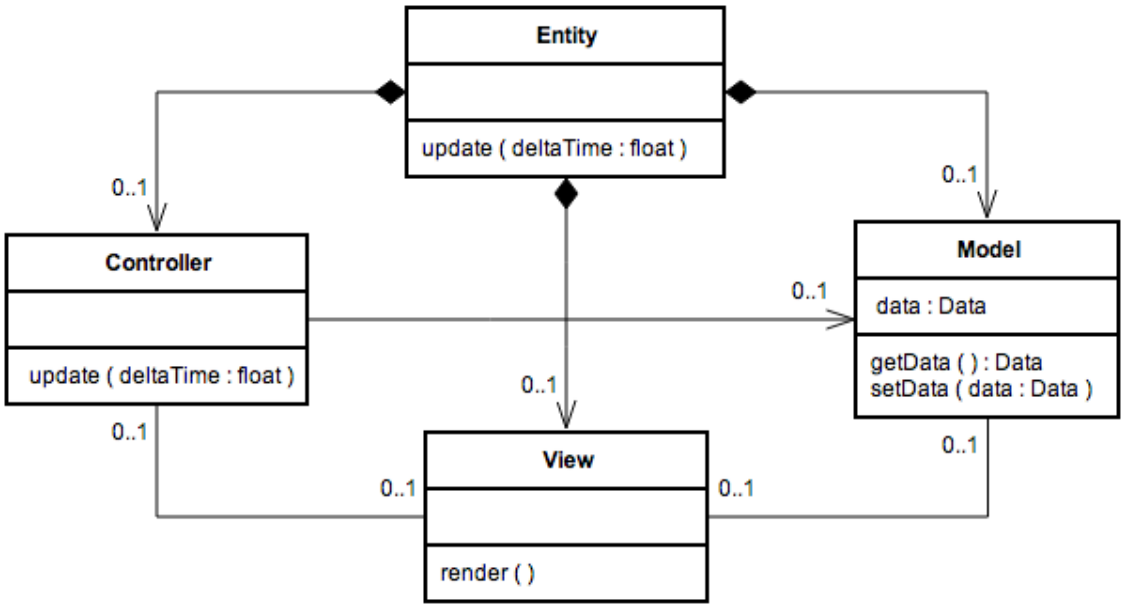
- **Malli:** Sisältää logiikan mallissa tarvittavien tietojen hakemiseen, päivittämiseen ja tallentamiseen. Pelissä malli voi olla vastuussa esimerkiksi kuvien lataamisesta.
- **Näkymä:** Mallin visuaalinen puoli. On vastuussa tarvittavien tietojen pyytämisestä mallilta, niiden prosessoimisesta ja näyttämisestä ruudulla.
- **Ohjaaja:** Toimii eräänlaisena linkkinä mallin, näkymän ja käyttäjän välillä. Vastaanottaa ja prosessoi käyttäjän syötteet ja muokkaa mallia tai näkymää sen mukaisesti.

(Paz 2013, 7-8.)

MVC-malli on tietyissä tilanteissa erityisen joustava, mikä toisaalta vaikeuttaa sen sisäistämistä. Esimerkkinä kuvassa 5 näkyvä nuoli näkymästä ohjaajaan. Näkymän vastuulla ei ole ohjaajan käskyttäminen, mutta se voi silti lähettää tietoja suoraan ohjaajalle ja vastaanottaa prosessoidun tuloksen. Tällaista toiminnallisuutta voidaan tarvita esimerkiksi pelin hahmojen repliikkien prosessoimisessa; malli sisältää repliikin tunnuksen, jonka näkymä lähettää ohjaajalle, joka palauttaa tunnusta vastaavan lokalisoidun merkkijonon.

MVC-mallin toimintamallien joustavuuden vastapainona on sen näkymän riippuvuussuhde mallista. Jotta näkymä voisi pyytää mallilta tarvittavat tiedot, on sen oltava tietoinen mallin julkisesta rajapinnasta ja käyttötavasta. Tätä voidaan yrittää kiertää luomalla geneerisiä rajapintoja, joita mallit toteuttavat, mutta yksi näkymä ei välttämättä sovellu kaikkien saman rajapinnan toteuttavien mallien kanssa käytettäväksi. (Koskimies & Mikkonen 2005, 142-144.)

Peliohjelmoinnissa MVC:n käyttö voi osoittautua hankalaksi, sillä mallin vastuualue on usein jaettu muiden luokkien kanssa. Malli ei myöskään voi korvata entiteettiä, sillä mallilla ei ole omistajuussuhdetta näkymän ja ohjaajan kanssa. Kaaviossa 7 on esitetty yksi tapa MVC-mallin käytöstä, joka soveltuu sekä lineaariseen hierarkiaan että puuhierarkiaan. Kaaviossa on edelleen Entity-luokka, joka kuitenkin nyt luo ja omistaa mallin, ohjaimen ja näkymän. Erona tavanomaiseen MVC-malliin on kuitenkin se, että kaikki kolme osaa ovat optionaalisia, sillä entiteeteillä ei välttämättä tarvitse olla visuaalista puolta (esim. tekoäly), eikä sen tarvitse ottaa vastaan käyttäjän syötettä ohjaajalla (esim. muuttumaton tekstikenttä). Jos entiteetillä ei ole mallia, ei tarvita myöskään näkymää. MVC:n eri osat on kaaviossa sisällytetty entiteettiin kompositiolla, jolloin entiteetin tuhoaminen tuhoaa myös ne. (Gamma, Helm, Johnson & Vlissides 2004, 166-167.)



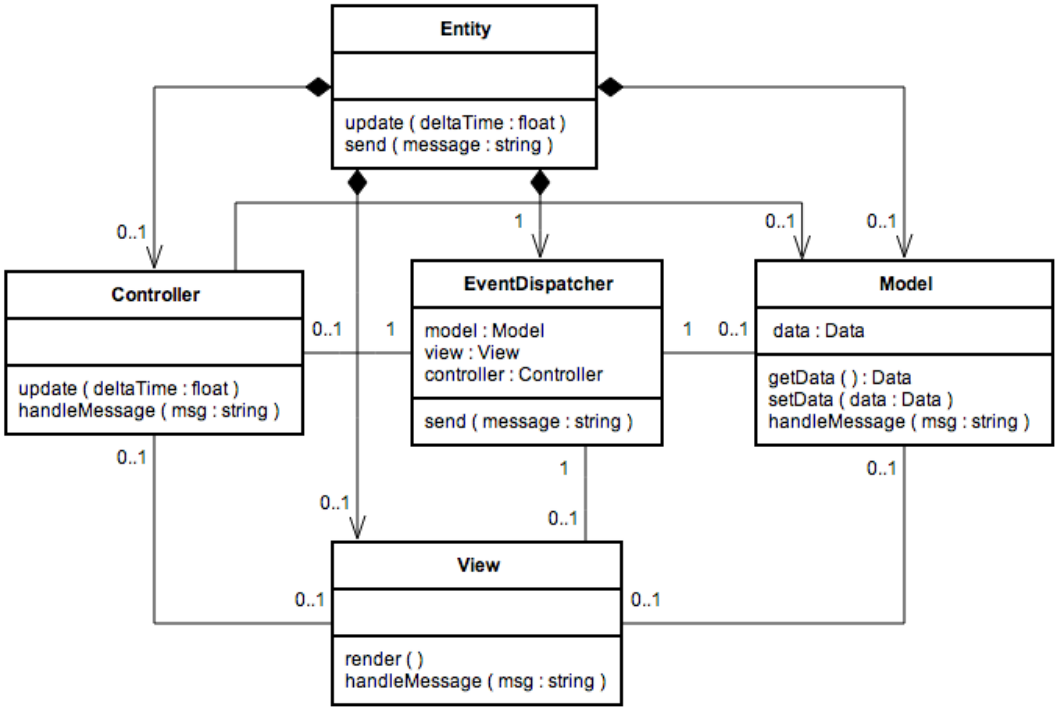
KAAVIO 7: Entiteetin toteutus MVC-mallilla

Kaavion 7 havainnollistamassa mallissa on kuitenkin yksi ongelma: instantiointijärjestys. MVC:n kaikki kolme osaa ovat enemmän tai vähemmän toisistaan riippuvaisia. Jos mallin luo ensimmäisenä ja näkymän toisena, on näkymän viite lähetettävä vielä jälkeenpäin mallille, kun taas mallin viitteen voi antaa suoraan näkymälle sen rakentajassa. Sama ongelma toistuu ohjaajan kanssa.

Myös tiedon välittäminen ulkopuolelta entiteetille hankaloituu, sillä mikä tahansa MVC:n osista voisi olla vastuussa viestin vastaanottamisesta. Ohjaaja voisi vastuualueidensa kannalta olla hyvä kandidaatti, mutta sitä ei välttämättä ole olemassa.

Tämä ongelma voidaan kiertää käyttämällä erillistä luokkaa viestien välittämiseen MVC:n eri osien välillä. Tätä käyttötarkoitusta varten luodaan EventDispatcher-luokka, jonka toiminnallisuutta on havainnollistettu kaaviossa 8. Entiteetti luo itselleen aina EventDispatcherin, jolle se välittää viitteet MVC-mallin eri osiin. Eri osat voivat edelleen keskustella suoraan toistensa kanssa, mutta mikäli ei haluta luoda riippuvuutta kutsumalla tiettyä metodia, voidaan sen sijaan lähettää viesti EventDispatcherille, joka välittää tiedon muille osille.

Entity-luokalle lisätään myös julkinen metodi viestien vastaanottamiseen. Sitä kutsutaan, kun halutaan lähettää viesti entiteetiltä toiselle. Entity ei itse käsittele viestiä, vaan ainoastaan välittää sen EventDispatcherille.



KAAVIO 8: Viestien välittäminen MVC-mallissa

Vaikka ongelma viestien välittämisessä onkin nyt korjattu, ei MVC tästä huolimatta sellaisenaan ole käytännöllinen peliohjelmoinnin tarpeisiin, eikä sitä siihen ole alun perin tarkoitettukaan. MVC ratkaisee vastuunjaon luokkien välillä, mutta aiheuttaa riippuvuuksia ja ns. *overhead*ia, eli ylimääräistä työtä sekä kehittäjältä että pelimoottorilta yksinkertaistenkin toimintojen suorittamiseen. Sovelluksen kannalta ylimääräiseksi työksi lasketaan kaikki prosessointiaika ja muistinkäyttö, joka ei suoraan edistä tehtävän suorittamista.

4.3 Model-View-Presenter

Model-View-Presenter eli MVP on MVC-mallin variaatio. Tässä luvussa tutustutaan erityisesti MVP-mallin passiiviseen toteutukseen. MVC-mallin tapaan MVP koostuu kolmesta eri osasta, joita on havainnollistettu kuvassa 6.

KUVA 6: Model-View-Presenter

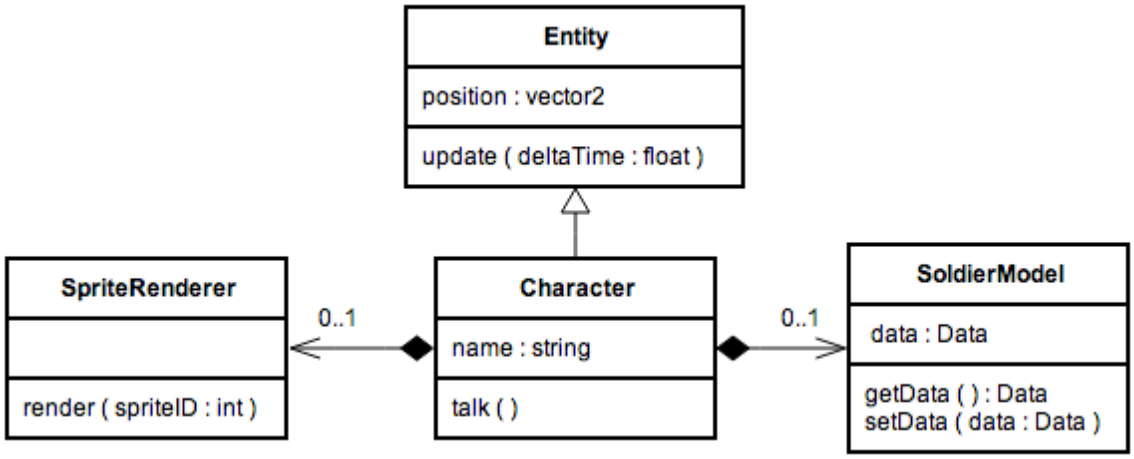
Osat ja niiden vastuualueet ovat:

- **Malli:** Rajapinta tietoon, jonka kautta kaikki tarvittava tieto on haettava. Voi joko toimia väylänä esim. tietokantaa tai sisältää tiedot itse. Vastuussa käyttäjän syötteiden validoinnista.
- **Näkymä:** Passiivinen näkymä, joka pelkästään renderöi mallilta saadut tiedot. Lukee myös käyttäjän syötteet ja lähettää ne eteenpäin esittelijälle. Näkymä on passiivinen, eikä se yleensä omaa tilaa.
- **Esittelijä:** Välikäsi näkymän ja mallin välissä. Kaikkien viestien on mentävä esittelijän läpi. Esittelijä on voitava korvata toisella ilman, että vaihdos vaikuttaa mallin ja näkymän toimintaan.

Suurin ero MVC-arkkitehtuuriin on mallin ja näkymän välisien riippuvuuksien poistaminen. Näkymä on myös täysin passiivinen, eikä ota osaa tiedon validointiin vaan seuraa ainoastaan esittelijän antamia ohjeita. MVP-malli on suunniteltu alun perin yksikkötestauksia silmällä pitäen, mihin sen eri osien itsenäisyys soveltuu hyvin. (Emmatty 2011.)

Kaaviossa 9 on kuvattu MVP-mallin yksi käyttökohde peliohjelmoinnissa. Kaaviossa on Entity-luokasta periytetty Character, joka toteuttaa hahmon toiminnallisuuden ja sisältää kaikille hahmoille keskeiset tiedot. Nämä ominaisuudet ja tiedot ovat siis sellaisia, jotka eivät erityisesti liity tietyn tyyppisiin hahmoihin, vaan joita kaikkien hahmojen voi olettaa tukevan. Character luokka luo kompositiolla SoldierModel-mallin ja SpriteRenderer-näkymän. Hahmo saa mallista hahmotyyppiinsä liittyviä tietoja ja

pystyy renderöimään itsensä ruudulle näkymänsä avulla. Vaikka esittelijän tulisikin toteuttaa kaikki entiteetin logiikan, voidaan hahmotyyppiin liittyviä ominaisuuksia sisällyttää malliin. Näin ollen esimerkiksi SoldierModel voi sisältää metodin *fight*, jolla tarjotaan hahmolle kyky taistella.



KAAVIO 9: Entiteetin toteutus MVP-mallilla

Hyvänä puolena MVP:ssä on overheadin vähäisyys verrattuna MVC:hen. Viestin välittäminen on myös selkeästi helpompaa, sillä esittelijä voi olla yksistään vastuussa viestien lähettämisestä ja vastaanottamisesta. Malli voi myös lähettää viestejä tilan muutoksista, joihin muiden entiteettien eri osat voivat reagoida, mutta sen ei ole tarpeen ottaa niitä vastaan lainkaan. MVP-malli tarjoaa myös hyvän mahdollisuuden yksikkötestaukseen, mikä on peliohjelmoinnissa melko harvinaista.

MVP-mallin hyvistä puolista huolimatta sen käytössä ilmenee erinäisiä ongelmia. Peliohjelmoinnissa mallin, näkymän ja itse logiikan ero on usein todella pieni, eikä näiden erittely eri luokkiin ole aina mahdollista. Näin käy esimerkiksi tilanteessa, jossa entiteetin näkymä vaikuttaa pelin logiikkaan muokkaamalla saman entiteetin tai sen mallin tilaa. Näkymän voi tarvita myös muokata muiden entiteettien käyttäytymistä. Tämän toiminnallisuuden voisi sijoittaa malliin, mutta tällöin näkymällä on oltava pysyvä tila, jota joko saman tai muiden entiteettien mallit lukevat. MVP-mallia voidaan kuitenkin ongelmistaan huolimatta soveltaa hyvin peliohjelmoinnissa, ottaen erinäisiä vapauksia ominaisuuksien toteuttamisen ja tiedon validoinnin vastuujaosta. (Potel 1996.)

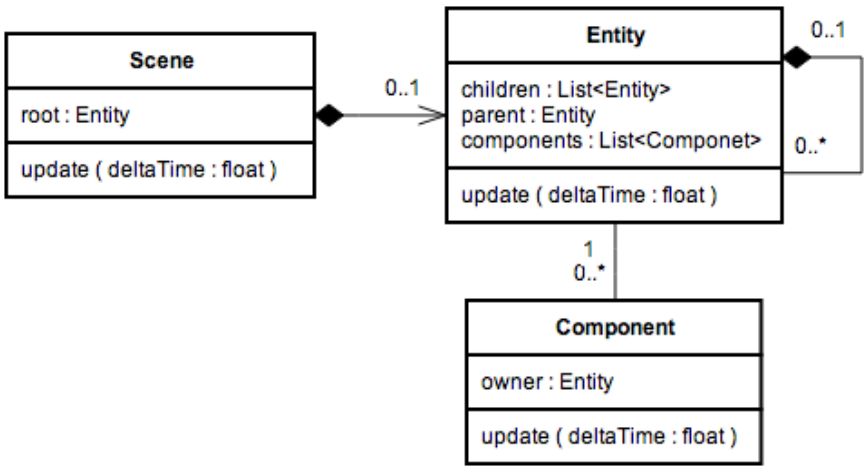
4.4 Entiteetti-komponentti

Entiteetti-komponentti-malli (eng. *entity-component*) on nykyään suosittu malli etenkin pelimoottoreissa, sillä se mahdollistaa pelin eri ominaisuuksien joustavan kehityksen. Mallia voi käyttää myös lineaarisen hierarkian kanssa, mutta tässä luvussa keskitytään esimerkin vuoksi pelkästään puuhierarkian käyttöön.

Entiteetti-komponentti-malli koostuu nimensä mukaisesti kahdesta eri käsitteestä: entiteeteistä ja komponenteista. Tämän mallin entiteettikäsite vastaa täsmälleen aiempien lukujen, entiteettejä. Tässä mallissa se kuitenkin sisältää ainoastaan kaikkein geneerisimmät tiedot, jotka on löydyttävä kaikilta entiteeteiltä. Nämä tiedot voivat käsittää entiteetin nimen, muunnosmatriisin ja sijainnin hierarkiassa. Näiden lisäksi entiteetti sisältää listan komponenteista, joista se on kompositiolla koostettu. Entiteetti itsessään ei siis sisällä erityistä toiminnallisuutta, vaan se koostetaan erilaisista komponenteista. Entiteetit kuitenkin saattavat tarjota metodeja hierarkiassa liikkumiseen ja komponenttien käsittelyyn. (Lord 2012.)

Komponentit tuovat entiteetteihin uusia ominaisuuksia ja toimintoja. Komponentteja ei erikoisteta periyttämällä, vaan yksittäiset ominaisuudet jaetaan erillisiksi komponenteiksi. Niiden tarkoituksena on siis toteuttaa joko yksi tai mahdollisimman rajattu kokonaisuus toimintoja.

Entiteetti-komponentti-mallissa on useimmiten myös kolmantena osana järjestelmä, joka vastaa pääosin luvussa 3.3 käytyjä globaaleja järjestelmiä. Tässä luvussa kuitenkin keskitytään entiteetti-komponentti-mallin yksinkertaistettuun muotoon, eli toteutukseen ilman järjestelmiä. Kaaviossa 10 on havainnollistettu tätä mallia. Siinä Scene-luokka omistaa yhden juurientiteetin eli Entity-luokan ilmentymän. Kaavion Entity-luokka sisältää listat alientiteeteistään ja komponenteistaan.



KAAVIO 10: Entiteetti-komponentti-malli

Esimerkiksi mallin käyttöön voi ottaa luvussa 3.2 (Puuhierarkia) kuvatun painonapin. Siinä painonappia vastaisi yksi entiteetti, jolla olisi neljä komponenttia. Kaksi näistä hoitavat molemmat yhden kuvan (taustakuva ja ikoni) renderöimisen ruudulle, kun taas kolmas komponentti renderöi painonapin tekstin. Entiteetti ei itse hoitaisi syötteen lukemista, vaan tätä varten siihen lisätään neljäs komponentti. Se lukee syötteen ja klikkauksen havaitessaan välittää siitä tiedon eteenpäin.

Entiteetti-komponentti-malli takaa ominaisuuksien joustavamman kehityksen, sillä ne voidaan eristää yksittäisiksi komponenteiksi. Tiedon hakeminen muista entiteeteistä tai komponenteista on myös helpompaa. Ongelmana tässä mallissa on kuitenkin komponenttien luokkien määrä monimutkaisissa ominaisuuksissa. Mikäli jonkin isomman ominaisuuden jokaisen yksittäisen toiminnon jakaa omaan komponenttiinsa, on ne jotenkin sidottava yhtenäiseksi kokonaisuudeksi, mikä yleensä tarkoittaa riippuvuussuhteiden luomista. Toinen mahdollinen ongelma on komponenttien päivitysjärjestys, mitä on kuvattu jo aiemmin luvussa 3.3 (Globaalit järjestelmät), jossa ankkuroinnin ja skaalauksen järjestys vaikuttaa lopputulokseen. Kolmas mahdollinen ongelma riippuu komponenttien viestintätavasta. Mikäli komponenttien on tarkoitus keskustella toistensa kanssa suoraan, antaa tämä vapaat kädet komponenteille vaikuttaa esimerkiksi entiteettihierarkiaan. Mikään ei siis estäisi komponenttia joko tarkoituksella tai vahingossa poistamasta kokonaista haaraa hierarkiasta. (West 2007.)

Entiteetti-komponentti-malli on ainakin Unity 3D:n käytössä, jossa useimmat ominaisuudet on toteutettu komponentteina. Pelin logiikan kehitys keskittyykin pääasiassa uusien komponenttien luomiseen ja niiden lisäämiseen entiteetteihin.

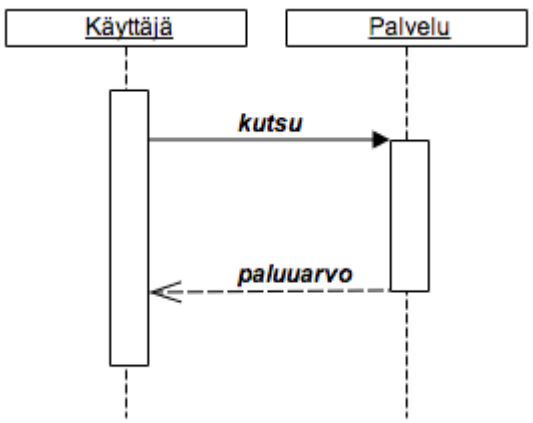
5 VIESTINTÄ

Entiteettien välisellä viestinnällä tarkoitetaan eri tapoja, joilla tietoa välitetään entiteetiltä toiselle. Nämä voivat olla joko käskyjä, tiedusteluita tai ilmoituksia. Viestit voivat kulkea sekä itse entiteettien, mutta myös niiden eri osien välillä. Viestintään on monia eri tapoja, eikä sovellukseen tarvitse valita vain yhtä, vaan oikea viestintätapa valitaan tilanteen mukaan.

5.1 Suorat kutsut

Suoralla kutsulla (eng. *direct call*) tarkoitetaan sitä, että ilmentymällä on viite toiseen ilmentymään, jonka kautta tämä kutsuu jotain sen julkista metodia. Tällainen on viitteen hankkimista lukuun ottamatta nopein tapa välittää tietoa, mutta luo luokkien välille riippuvuussuhteita. (Hietanen 2004, 191.)

Suoria kutsuja käytetään etenkin silloin, kun luokan on loogista käyttää kohdetta ilman välikäsiä. Esimerkiksi entiteetti voi kutsua suoraan komponenttejaan, sillä ne ovat osa samaa kokonaisuutta. Suoria kutsuja käytetään myös silloin, kun kutsun kohde tiedetään varmaksi ja viesti halutaan välittää vain tuolle yhdelle kohteelle. Suorat kutsut eivät erityisesti suosi mitään arkkitehtuuria, vaan niitä käytetään tilanteen mukaan. Kaaviossa 11 on esitetty yksinkertainen esimerkki suoran kutsun toiminnasta kahden luokan välillä. Siinä käyttäjä kutsuu palvelun julkista metodia, joka palauttaa takaisin paluuarvon. (Nystrom 2013.)

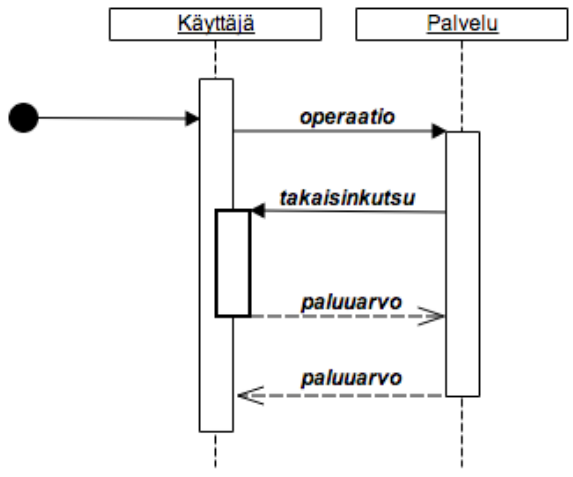


KAAVIO 11: Suoran kutsun toiminta

5.2 Takaisinkutsut

Takaisinkutsuilla (eng. *callback*) tarkoitetaan suoritettavan ohjelmakoodin lähettämistä parametrina toisen luokan julkiseen metodiin. Parametrin tietotyyppi riippuu ohjelmointikielestä ja toteutustavasta. Kun luokka kutsuu toisen luokan metodia antaen parametrina tällaisen muuttujan, jota tuo toinen luokka vastaavasti kutsuu, on kyseessä takaisinkutsu. Takaisinkutsu voidaan suorittaa joko synkronisesti heti tai asynkronisesti myöhemmin, kun jotkin ehdot täyttyvät. Takaisinkutsujen perimmäisenä tarkoituksena on välttää turhien riippuvuussuhteiden luomista luokkien välille.

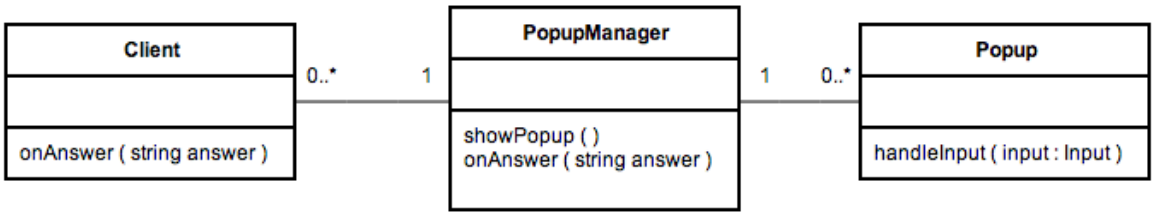
Ohjelmointikielestä riippuen takaisinkutsu voidaan toteuttaa esimerkiksi raakaosoittimella, eli suoralla muistiosoitteella metodiin, tai funktoreilla, eli olioilla, joita voidaan kutsua aivan kuin ne olisivat tavallisia metodeja. Monet kielet myös tarjoavat mahdollisuuden luoda anonyymeja metodeja ns. lambda-toiminnallisuutta käyttäen, joita ei ole sidottu tunnistajaan, eli luokkaan tai muuttujaan. Kaaviossa 12 on esitetty sekvenssinä takaisinkutsun toiminta käyttäjän ja palvelun välillä. Kaaviossa käyttäjä aloittaa operaation kutsumalla palvelun julkista metodia. Operaation aikana palvelu kutsuu jotain käyttäjän julkista metodia ja saa mahdollisesti takaisin paluuarvon. Lopulta palvelun operaatio loppuu, palvelu antaa takaisin paluuarvon ja ohjelma jatkaa suorittamista käyttäjässä. (Deitel & Deitel 2013, 16.4.)



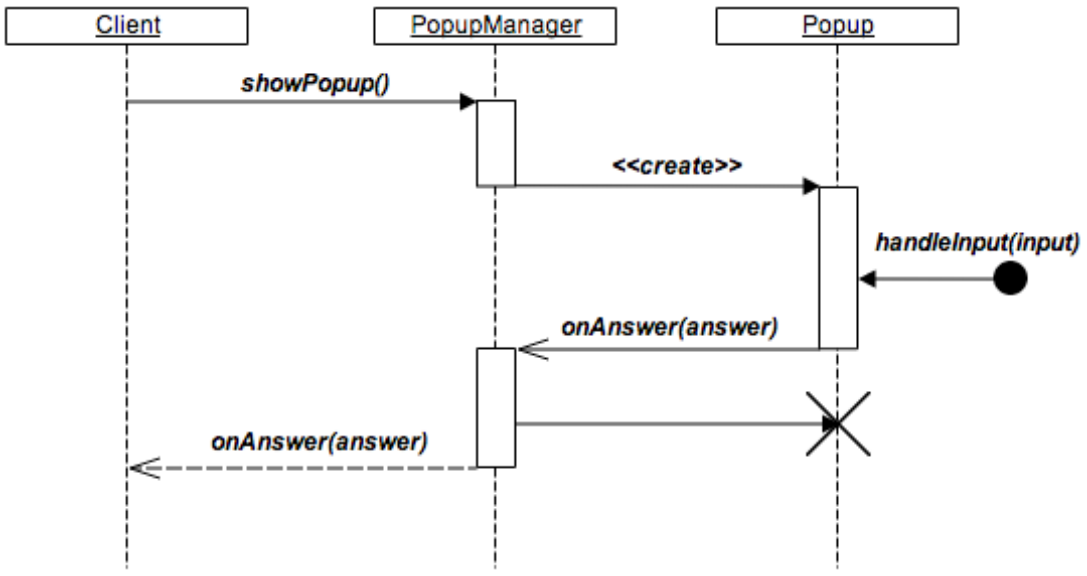
KAAVIO 12: Takaisinkutsu sekvenssikaaviona

Esimerkki takaisinkutsujen käyttökohteesta peliohjelmoinnissa voisi olla ponnahdusikkunoista vastaavan järjestelmän toiminta. Kun tälle tulee pyyntö ponnahdusikkunan näyttämisestä, luo se uuden ilmentymän ponnahdusikkunasta,

populoi sen sisällön ja asettaa sen ruudulle näkyviin. Ponnahdusikkunoissa on usein painikkeita, joita klikkaamalla pelaaja voi vastata ponnahdusikkunan sisältämään kysymykseen tai ilmoitukseen. Ponnahdusikkuna voi välittää vastauksen suoraa ponnahdusikkunoista vastaavalle järjestelmälle, mutta tämä sen sijaan ei voi välittää sitä eteenpäin ulkopuolelle suoralla kutsulla. Muutoin järjestelmällä olisi turhia riippuvuussuhteita moniin muihin luokkiin. Kaaviossa 13 on havainnollistettu tällaista toiminnallisuutta suoria kutsuja käyttäen. Siinä PopupManager vastaa ponnahdusikkunoista vastaavaa järjestelmää, Popup kaikkien ponnahdusikkunoiden kantaluokkaa ja muut luokat pyyntöjen lähettäjiä. Kaaviosta nähdään riippuvuussuhteet, joita luokkien välille muodostuu. Kaaviossa 14 on esitetty kaavion 13 toiminta sekvenssinä PopupManagerin näkökulmasta. PopupManager sisältää *showPopup*-metodin, jota Client kutsuu. PopupManager luo uuden ilmentymän Popup-luokasta, joka jää odottelemaan käyttäjän syötettä. Kun syöte on saatu, lähettää Popup vastauksen asynkronisesti PopupManagerille kutsumalla sen *onAnswer*-metodia, josta vastaavasti kutsutaan Client-luokan *onAnswer*-metodia.

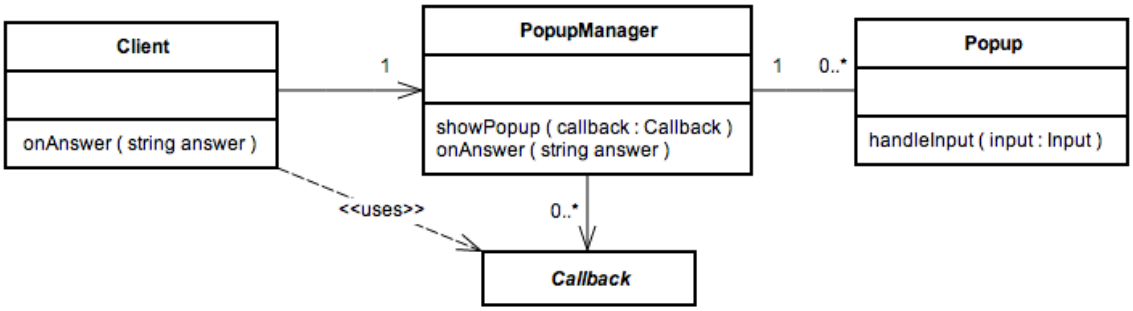


KAAVIO 13: Suorista kutsuista aiheutuvat riippuvuussuhteet

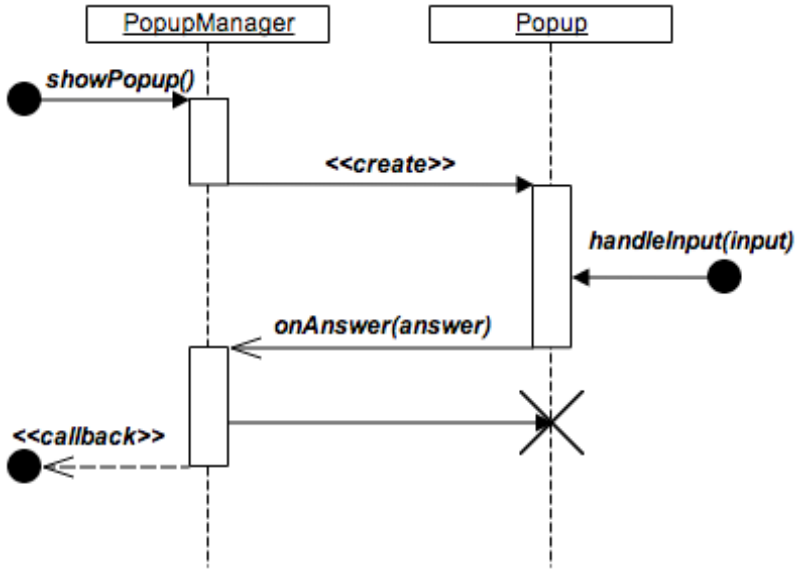


KAAVIO 14: PopupManagerin toiminta suorilla kutsuilla

Suoran kutsun sijaan järjestelmää käyttävät luokat välittävät pyyntönsä mukana viitteen metodiin, johon lopullinen ponnahdusikkunan välittämä vastaus lähetetään. Näille metodeille on määritelty tarkka otsikko, joka metodeilla on oltava. Järjestelmä ei siis tarkalleen tiedä, mihin vastaus lähetetään, eikä riippuvuussuhdetta näin ollen synny. Kaaviossa 15 on havainnollistettu takaisinkutsun käytöstä. Sen toimintaperiaate on muuten identtinen kaavion 13 kanssa, mutta riippuvuussuhde PopupManagerista Clientiin on korvattu takaisinkutsuolion käytöllä. Kaavion 15 toiminta on esitetty sekvenssinä PopupManagerin näkökulmasta kaaviossa 16. Tästä erityisesti huomataan, että pyynnön lähettäjää ei enää tarvitse näyttää, sillä siitä ei PopupManagerin näkökulmasta tiedetä.



KAAVIO 15: Riippuvuussuhteiden välttäminen takaisinkutsuilla



KAAVIO 16: PopupManagerin toiminta takaisinkutsuilla

Takaisinkutsujen tarkoituksena on tehdä toiminnoista joustavia, helpommin skaalautuvia ja muista riippumattomia. Kun useat erilaiset luokat käyttävät samaa

toimintoa, joka vaatii tiedon välittämistä asynkronisesti, tulisi miettiä, voisiko sen toteuttaa takaisinkutsuilla.

Huonona puolena takaisinkutsujen käytössä on kuitenkin mahdollisien ongelmakohtien etsiminen etenkin silloin, kun ne on toteutettu raakaosoittimilla. On voitava jotenkin varmistaa, että takaisinkutsun kohde on vielä olemassa, tai muutoin takaisinkutsun suorittaminen aiheuttaa muistivirheen, mikä voi mahdollisesti aiheuttaa odottamattomia seurauksia.

5.3 Entiteetin tilan muuttaminen

Keskustelu entiteetin tilaa muuttamalla on keino mahdollistaa luokkien välinen keskustelu ilman minkäänlaisia riippuvuussuhteita. Tämä keskustelutapa soveltuu modulaarisiin rakenteisiin, joissa entiteetti on koostettu kompositiolla useista eri luokkien ilmentymistä.

Hyvä esimerkki tällaisesta keskustelutavasta olisi tilanne, jossa kompositiolla koostettu entiteetti sisältää kaksi komponenttia: Movement ja Renderer. Esimerkissä entiteetti sisältää tiedon sijainnistaan. Movement-komponentti lukee käyttäjän syötteitä ja muuttaa entiteetin sijaintia vastaavasti. Renderer-komponentti sen sijaan lukee entiteetistä sijainnin ja piirtää entiteettiä vastaavan kuvan ruudulle sen sijaintia vastaavaan kohtaan. Nämä komponentit eivät keskustele keskenään suoraan, vaan muokkaamalla entiteetin tilaa; Movement asettaa sijainnin ja Renderer lukee sen. Näin nämä kaksi komponenttia ovat toisistaan näennäisesti riippumattomia.

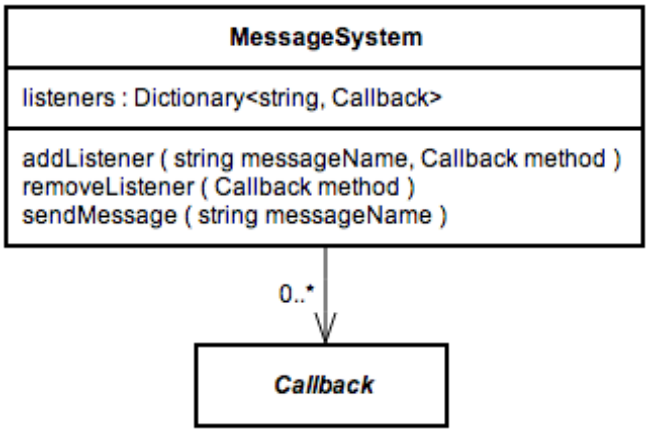
On kuitenkin huomattava, että eräänlainen riippuvuussuhde komponenttien välillä kuitenkin on, sillä niiden suoritusjärjestys vaikuttaa lopputulokseen. Mikäli Renderer lukee sijainnin ennen kuin Movement asettaa sen, näytettäisiin ruudulla aina edellinen sijainti. Ainoastaan entiteetin pysyessä paikoillaan kahden päivityskerran ajan, näytettäisiin ruudulla oikea tilanne.

Tämä keskustelumuoto vaatii myös tietojen sisällyttämistä entiteetteihin, jolloin tietojen olisi entiteettien uudelleenkäyttöisyyttä ajatellen pysyttävä mahdollisimman geneerisinä. Jos komponentti asettaisi entiteetille tiedon hahmon sanomasta repliikistä,

jonka toinen komponentti renderöi tekstinä ruudulle, ei entiteettiä voisi enää järkevästi käyttää kuvaamaan ympäristöä, kuten puita ja kiviä. (Nystrom 2013.)

5.4 Keskitetty viestijärjestelmä

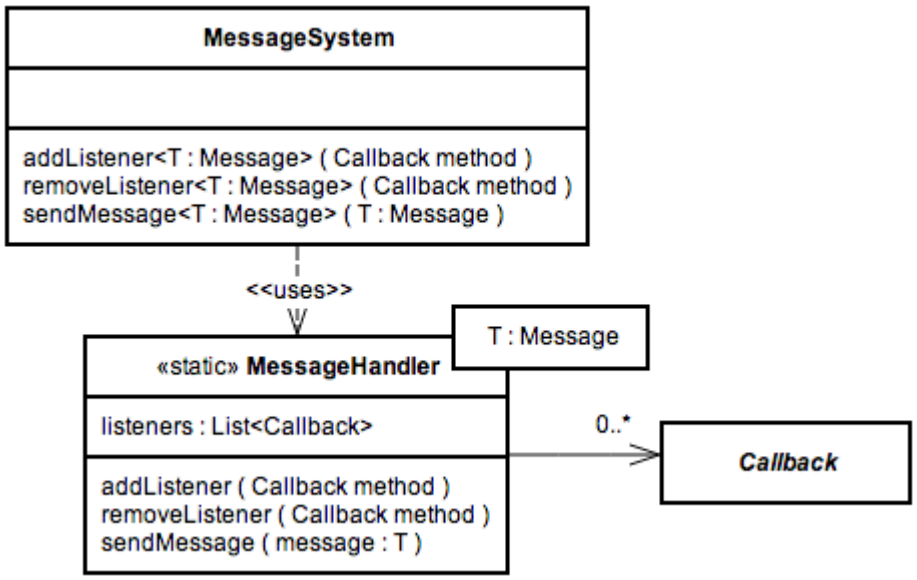
Monet pelimoottorit toteuttavat ison osan entiteettien välisestä keskustelusta keskitetyillä viestijärjestelmillä. Viestijärjestelmä on käytännössä luokka, jolle viestit annetaan toimitettavaksi ilman tietoa vastaanottajasta. Muut viestistä kiinnostuneet luokat rekisteröivät itsensä viestijärjestelmään tämän viestin kuuntelijoiksi. Viestijärjestelmä tarvitsee siis metodit viestin lähettämiseen ja kuuntelijoiden lisäämiseen sekä poistamiseen. Tällaiset viestijärjestelmät toteutetaan yleensä takaisinkutsuilla, joissa viestijärjestelmän ei tarvitse tietää, mistä viesti lähetettiin tai mille luokalle se on välitettävä. Se ainoastaan pitää muistissaan osoitteita, joihin tietynlaiset viestit on lähetettävä. Viestijärjestelmä kuitenkin vaatii, että viestejä vastaanottavien metodien otsikot ovat oikeanlaisia. Kaaviossa 17 on kuvattu tällaisen viestijärjestelmän rakennetta.



KAAVIO 17: Viestijärjestelmä ilman viestiparametreja

Kaavion 17 viestijärjestelmä ei kuitenkaan salli parametrien välittämistä luokkien välillä, vaan ainoastaan tietää viestien nimet. Jotta viestien mukana voitaisiin lähettää mitä tahansa viestejä, on viestien oltava samasta kantaluokasta periytettyjen luokkien ilmentymiä. Viestejä vastaanottavien metodien otsikoiden on edelleen oltava oikeanlaisia (ottavat vastaan yhden ko. kantaluokan tyyppiä olevan parametrin). Koska viestiä vastaa tällaisessa järjestelmässä luokka, voidaan kuuntelijat rekisteröidä viestien nimien sijaan tietotyypeillä. Tällöin vältetään kirjoitusvirheistä johtuvat virheet, sillä

tietotyypin väärinkirjoittamisesta aiheutuvat virheet löydetään jo käännösvaiheessa. Kaaviossa 18 on kuvattu tällaisen viestijärjestelmän rakennetta. Siinä viestijärjestelmän metodit on toteutettu malleina, joita kutsuessa on annettava tyyppiparametrina viestiä vastaavan luokan tyyppi. Viestijärjestelmä ei itse käsittele kutsuja, vaan ainoastaan välittää ne tyyppiparametria vastaavalle käsittelijälle. Käsittelijät ovat versioita staattisesta MessageHandler-malliluokasta, jotka on luotu jo sovelluksen käännösvaiheessa. Käsittelijät ovat vastuussa kuuntelijoiden lisäämisestä ja poistamisesta sekä viestien käsittelystä.



KAAVIO 18: Viestijärjestelmä viestiparametreilla

Ohjelmakoodissa 4 on näytetty, kuinka kaavion 18 mukaisen viestikäsittelijän voi toteuttaa C#-kielellä. Ohjelmakoodi sisältää siis metodit *addListener*, *removeListener* ja *sendMessage*. Toisin kuin viestijärjestelmässä, näitä ei ole tarpeen toteuttaa metodimalleina, sillä luokka itsessään on jo malli, joka ottaa vastaan tyyppiparametri *T*:n. Luokalla on myös tietojäsen *listeners*, joka on lista siihen lisätyistä takaisinkutsuolioista. C#:sta löytyy valmiina *System.Delegate*-luokasta periytetty *System.Action*<*T*>, joka kapseloi sisäänsä takaisinkutsun metodiin, joka ottaa vastaan yhden parametrin, eikä palauta paluuarvoa.

```
using System;
using System.Collections.Generic;

static public class MessageHandler<T> where T : Message
{
    static private List<Action<T>> listeners;

    static public void addListener( Action<T> callback )
    {
        if ( listeners == null )
            listeners = new List<Action<T>>();

        listeners.Add( callback );
    }

    static public void removeListener( Action<T> callback )
    {
        listeners.Remove( callback );
    }

    static public void sendMessage( T message )
    {
        foreach ( Action<T> action in listeners )
        {
            action( message );
        }
    }
}
```

OHJELMAKOODI 4: MessageHandler C#-kielellä

Kuten ohjelmakoodista 4 huomataan, ei viestikäsittelijän tarvitse tietää muuta kuin viestin tyyppi. Etuna on myös se, että viestijärjestelmänkään ei tarvitse tietää kuin viestin tyyppi voidakseen välittää kutsun eteenpäin. Ohjelmakoodiin 5 on toteutettu kaavion 18 viestijärjestelmän *sendMessage*-metodi.

```
public void sendMessage<T>( T message )
{
    MessageHandler<T>.sendMessage( message );
}
```

OHJELMAKOODI 5: MessageHandlerin käyttö MessageSystemissä

Viestijärjestelmän toteuttamistavasta huolimatta sen hyöty on sama: keino lähettää viestejä kahden toisistaan täysin tietämättömän luokan välillä ilman riippuvuussuhteiden muodostumista. Viestijärjestelmää on hyvä käyttää yhdessä muiden viestintätapojen kanssa. Kun halutaan lähettää yksi viesti entiteetiltä toiselle, voidaan käyttää suoria kutsuja, mutta kun viestin vastaanottajista ei ole tietoa, lähetetään viesti viestijärjestelmän kautta. Viestijärjestelmiä käytetäänkin yleensä ilmoittamaan erinäisistä tapahtumista, joihin voi mahdollisesti useampikin luokka reagoida.

Ongelmana keskitettyjen viestijärjestelmien käytössä on pelin logiikan monimutkaistuminen. Koska viestejä voi kuunnella mikä luokka tahansa, ei aina voida olla varma tapahtumien aiheuttamista seurauksista. Ohjelmoijien olisikin hyvä pyrkiä dokumentoimaan erilaiset viestityypit ja niiden lähettäjät sekä vastaanottajat, jolloin tapahtumaketjujen seuraaminen on helpompaa.

6 YHTEENVETO

Kuten huomasin pelimoottoriani tehdessäni, ei erilaisista toteutustavoista välttämättä huomaa niiden heikkouksia pelkästään teorian perusteella. Usein idea oli toteutettava pelimoottoriin ja käytettävä sitä vähintään yhdessä käytännön esimerkissä. Peliohjelmointi eroaa perinteisestä sovellusohjelmoinnista huomattavasti, sillä siinä esimerkiksi näkymien ja mallien välinen ero ei aina ole selkeä. Näkymä saattaa vaikuttaa pelin logiikkaan tai mallin on pystyttävä kontrolloimaan toista entiteettiä. Pelien vaatimukset voivat siis rikkoa pelimoottoriin suunnitellun arkkitehtuurin rajoja, jolloin on pyrittävä luomaan kompromissiratkaisuja.

Opinnäytetyö seuraa siis suurin piirtein oman pelimoottorini iteraatioita, joita olen tämän opinnäytetyön valmistumiseen mennessä tehnyt yhteensä kuusi. Aloitin opinnäytetyöni tapaan yksinkertaisimmista toteutustavoista. Ensimmäinen iteraatio esimerkiksi käytti yhtä keskitettyä pääsilmukkaa, lineaarista hierarkiaa ja renderöinti oli hoidettu globaalilla piirtolistalla, johon entiteetit lisäsivät kuvia ja jota renderöijä prosessoi. Entiteettien rakenne perustui myös täysin periyttämiseen eikä toimintoja ollut lähes lainkaan ulkoistettu. Tämän toteutuksen huono puoli oli helppo huomata: pientenkin muutoksien tekeminen vei päiviä. Luokat myös sisälsivät tietoja ja ominaisuuksia, jotka eivät loogisesti liittyneet niihin mitenkään.

Opiskeltuani suunnittelu- ja arkkitehtuurimalleja, siirryin seuraavissa iteraatioissa askel kerrallaan kohti järjestelmällisempiä ja suunnitelmallisempia toteutustapoja. Tutustuin olioiden koostamiseen kompositiolla useammasta oliosta MVC- ja MVP-malleilla. Näistä siirryin komponenttien käyttöön kun vuoden 2013 keväällä tutustuin Unity 3D –pelimoottoriin ja sitä kautta ECS-malliin (*Entity-Component-System*), jota Unity 3D hyödyntää lähes kaikessa. Siirryin käyttämään samaa mallia myös omassa pelimoottorissani, tehden kuitenkin pieniä muutoksia, jotka koin pelimoottorin toteutuksen tässä vaiheessa tarpeellisiksi. Poistin järjestelmät kyseisestä ja mallista siirsin niihin liittyvän perustoiminnallisuuden suoraan entiteetteihin. Tämä teki nopean kehityksen mahdolliseksi, sillä uusia komponentteja tehdessä ei ollut enää tarvetta pitää myös järjestelmiä ajan tasalla.

Kuten pelimoottorini iteraatioissa, myös opinnäytetyössä päädyin päälukujen lopuissa puuhierarkiaan, entiteetti-komponentti-malliin ja viestijärjestelmiin. Järjestystä ei ole valittu sattumalta, vaan pelimoottorini hyödyntää tämän opinnäytetyön valmistumishetkellä pääosin juuri näitä toteutustapoja. Näihin olen päätynyt punnitsemalla niiden vahvuuksia ja heikkouksia. Olen suosinut nopeuden ja testattavuuden sijaan helppokäyttöisyyttä ja joustavuutta.

Suurinta osaa peliohjelmoinnista oppimaani olen päässyt soveltamaan käytännössä sekä omaa pelimoottoriani tehdessä että peliohjelmoijan työtehtävissäni Rovio Entertainment Oy:llä, ja juuri näihin käytännön kokemuksiin suurin osa päätelmistäni perustuukin. Kuten missä tahansa tutkimuksessa, myös tässä päätelmiin ovat saattaneet vaikuttaa aiemmat sekä onnistuneet että epäonnistuneet yritykset erinäisten arkkitehtuurimallien käytössä, jolloin aiheesta ei voi antaa täysin objektiivista näkökulmaa.

Objektiivisen näkökulman antamisessa huomasin ongelmia myös lähdemateriaalien suhteen. Peliohjelmoinnista ei ole kirjoitettu kovinkaan montaa kirjaa, jotka antaisivat yleiskuvan peliohjelmoinnista korkealla tasolla. Useimmat aiheesta kirjoitetut kirjat kertovat eri asioiden toteuttamisesta tietyissä kehitysympäristöissä, joissa pätevät rajoitukset tai vapaudet eivät välttämättä päde muissa ympäristöissä. Lähdemateriaaleja valitessani olenkin pyrkinyt ottaa selvää kirjoittajan taustasta ja yrittänyt sen perusteella päätellä lähteen luotettavuuden. Esimerkkinä Robert Nystromin kirjoittama *Game Programming Patterns*. Kirja on keskeneräinen, mutta kirjoittajalla on tietotekniikan koulutus ja pitkä kokemus ohjelmistojen kehittämisestä. Myös suosittujen kirjasarjojen laitoksia pidin luotettavina lähteinä, kuten kirjaa *Design Patterns, Elements of Reusable Object-Oriented Software*, joka kuuluu Addison-Wesley-kirjasarjaan, jonka kirjoja on julkaistu vuodesta 1942 alkaen. Kirjan kirjoittajat ovat olleet kehittämässä monia tunnettuja projekteja kuten SmallTalk, JUnit, Eclipse ja Java Development Tools. Kirjan esipuheen kirjoittanut Grady Booch on yksi UML:n (*Unified Modeling Language*) kehittäjistä, jota tämänkin opinnäytetyön kaavioissa olen käyttänyt.

LÄHTEET

Deitel, P. & Deitel, H. 2013, 9. painos. C++ How To Program.

Eberly, D. 2004. 3D Game Engine Architecture.

Emmatty, J. 11/2011. Differences between MVC and MVP for Beginners. Luettu 13.11.2013. www.codeproject.com/Articles/288928/Differences-between-MVC-and-MVP-for-Beginners

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. 2004, 28. painos. Design Patterns, Elements of Reusable Object-Oriented Software.

Gregory, J. 6/2009. Game Engine Architecture.

Hietanen, P. 2004, 8. painos. C++ ja olio-ohjelmointi.

Koskimies, K. & Mikkonen, T. 2005. Ohjelmistoarkkitehtuurit.

Laine, H. (Helsingin yliopisto). 8/1996. Oliosanasto. Luettu 1.12.2013. www.cs.helsinki.fi/u/laine/oliosanasto

Lord, Richard. 1/2012. What is an entity system framework for game development? http://www.richardlord.net/blog/what-is-an-entity-framework

Nystrom, R. (Electronic Arts). 2013. Game Programming Patterns. Luettu 4.10.2013. gameprogrammingpatterns.com

Paz, J. R. G. 2013. Beginning ASP.NET MVC 4.

Potel, M. 1996. MVP: Model-View-Presenter, The Taligent Programming Model for C++ and Java. http://www.wildcrest.com/Potel/Portfolio/mvp.pdf

Sivak, S. 8/2009. What Is A Game Engine. Luettu 15.12.2013. slideshare.net/sjsivak/what-is-a-game-engine

Ward, J. 4/2008. What is a Game Engine? Luettu 13.12.2013. gamecareer-guide.com/features/529/

West, M. 1/2007. Evolve Your Hierarchy. Luettu 25.11.2013. cowboyprogram-ming.com/2007/01/05/evolve-your-heirachy/

Zechner, M. & Green, R. 2011. Beginning Android 4 Games Development.